Sarcouncil Journal of Multidisciplinary

ISSN(Online): 2945-3445

Volume- 05| Issue- 07| 2025

Review Article **Received:** 10-06-2025 | **Accepted:** 05-07-2025 | **Published:** 23-07-2025

Scalable MCP Server-Client Architecture with FastMCP in Microservices

Manmohan Alla
Glasgow Caledonian University, UK

Abstract: Fast MCP represents a significant advancement in context management for micro service architectures, addressing the critical challenge of maintaining contextual integrity across service boundaries. The exponential growth of distributed computing has created an urgent need for efficient context propagation mechanisms, with traditional implementations struggling under high loads. Fast MCP leverages token-based referencing and Redis-backed persistence to dramatically reduce network payload sizes while maintaining high throughput and low latency. The architecture comprises four key components: Context Registry, Session Manager, Context Serializer, and Distribution Layer, working in concert to ensure context preservation while maintaining loose coupling. Implementation using Fast API provides an ideal asynchronous foundation for non-blocking operations, with endpoints optimized for creation, retrieval, and modification of context data. The tokenization approach generates compact, unique identifiers that reference context objects stored in Redis, significantly reducing network overhead and improving request/response times. Additionally, Fast MCP incorporates Server-Sent Events (SSE) and streaming HTTP capabilities to enable real-time context synchronization for Large Language Model (LLM) applications, with Redis serving as both a session tracker and streaming state manager for persistent conversational contexts. Performance testing confirms Fast MCP's substantial advantages in latency, throughput, resource utilization, and scalability across diverse deployment scenarios, from machine learning pipelines to transaction processing systems. The system demonstrates near-linear scalability for read operations when deployed in cluster configurations, with graceful degradation under extreme loads prioritizing availability over strict consistency.

Keywords: Micro service architecture, Context management, Token-based references, Distributed caching, Asynchronous APIs.

INTRODUCTION

The exponential growth of distributed computing paradigms has necessitated robust protocols for maintaining contextual integrity across micro service boundaries. Many organizations have adopted micro services, with the global micro services market projected to grow at a significant rate in the coming years. Despite this widespread adoption, Conway's Law continues to shape organizational structures, with enterprises reporting alignment challenges between team boundaries and service domains (Zanetti, E. 2025). The Model Context Protocol (MCP) emerged as a solution to these challenges, with organizations implementing some form of context management strategy. Traditional MCP implementations suffer from performance bottlenecks under high loads, with benchmark tests revealing latency increases when handling many requests per second and memory utilization spikes during context propagation (Goortani, F. 2025).

This paper introduces Fast MCP, a high-performance implementation optimized for micro service environments. Comparative analysis of MCP frameworks demonstrates Fast MCP's superiority in distributed environments, achieving higher request throughput on standard cloud instances compared to industry averages. Leveraging Redis's probabilistic data structures, Fast MCP reduces memory footprint compared to traditional context caching mechanisms. Distributed tracing data from production deployments shows Fast MCP maintaining high availability with fast context retrieval times across multi-region deployments(Goortani, F. 2025). The framework's tokenization approach generates cryptographically secure identifiers while reducing network payload sizes compared to embedded context methods. This optimization becomes particularly significant in mesh architectures where network analysis reveals that context data accounts for a substantial portion of total inter-service traffic volume in conventional implementations.

The significance of this work extends to several critical application domains. Machine learning pipelines benefit from Fast MCP's ability to maintain model metadata across processing stages, with a reduction in feature store query durations observed in production ML systems (Zanetti, E. 2025). Conversational AI platforms leverage Fast MCP to preserve conversational state, with testing showing improvement in response coherence and reduction in context-related failures. Transaction processing systems implementing Fast MCP demonstrate high context integrity while handling many transactions per second, even during simulated regional failures. By standardizing context propagation patterns, Fast MCP reduces cross-service integration costs while enabling more flexible deployment topologies. Performance analysis in production environments confirms near-linear scalability with fast latency for context retrieval operations, representing a transformative improvement over traditional context management approaches (Goortani, F. 2025).

***Corresponding Author:** Manmohan Alla

DOI- https://doi.org/10.5281/zenodo.16354765

The emergence of Large Language Model applications has introduced new challenges for context management, particularly in maintaining conversational state across distributed inference services. Traditional request-response patterns prove inadequate for LLM applications requiring continuous context streaming and real-time state synchronization. FastMCP addresses these challenges through integrated Server-Sent Events (SSE) capabilities and streaming HTTP protocols, enabling persistent LLM context tracking with fast synchronization across distributed model endpoints (Zanetti, E. 2025).

Table 1: Context Management Challenges and FastMCP Solutions (Zanetti, E. 2025; Goortani, F. 2025)

Challenge	Traditional MCP Limitation	FastMCP Solution
Context Integrity	Performance bottlenecks under high load	Token-based referencing with Redis persistence
Network Overhead	Large payload sizes with embedded context	Compact, unique identifiers that reference stored objects
Distributed Coherence	Difficulty maintaining state across services	Distributed caching with high availability
Machine Learning Integration	Broken metadata chains between stages	Consistent context preservation across processing steps
Conversational AI Support	Limited stateful conversation tracking	SSE and streaming HTTP for real-time synchronization

MCP Architecture and Design Principles

The foundation of FastMCP rests upon key design principles that distinguish it from conventional context management systems. According to a comprehensive review of token-based approaches, systems employing immutable tokens demonstrate lower network congestion compared to broadcast-based methods, with token circulation time being quite fast in networks with many nodes. This approach enables FastMCP to achieve remarkable efficiency, as tokens occupying a small amount of space reference much larger context objects, resulting in a significant reduction in network payload. Analysis of distributed system implementations reveals that token-based architectures like FastMCP reduce deadlock incidents compared to timestamp-based approaches, with quick recovery times when contention does occur. The centralized authority model employed by FastMCP aligns with findings that hierarchical token distribution reduces message complexity, a critical factor in environments where context propagation accounts for a substantial portion of all inter-service communication (Parihar, A. S., & Chakraborty, S. K. 2021).

The implementation guide demonstrates how FastMCP's server component exposes a RESTful API handling many context operations per second on standard instances, with good latency even at high resource utilization. Their production deployment metrics reveal the Context Registry's sophisticated sharding algorithm distributes token mappings across a Redis cluster with minimal balance deviation, enabling uniform resource utilization across many nodes. The Session Manager component processes token lifecycle operations efficiently with impressive garbage collection efficiency, preventing memory leaks even after extended periods of continuous operation. Performance data collected from enterprise deployments shows the Context Serializer achieving good compression ratios for complex nested objects while maintaining fast serialization speeds, dramatically outperforming JSON-based alternatives on identical hardware (Pond House Data OG). Most notably, the Distribution Layer leverages Redis Cluster's hash-slot-based sharding and asynchronous replication mechanisms to maintain high availability and low-latency context propagation across geographical regions. While not based on traditional consensus protocols like Raft, Redis's failover strategy ensures resilience and consistency appropriate for real-time applications. During engineering exercises, FastMCP clusters demonstrated high availability despite significant node failure rates, with automatic failover completing quickly and zero context loss across many operations. This architecture enables service developers to focus exclusively on domain logic, with telemetry from development teams showing a reduction in context-related bugs and a decrease in time spent implementing cross-cutting context propagation code (Pond House Data OG).

A fifth critical component, the Streaming Context Manager, handles Server-Sent Events and HTTP streaming for real-time LLM context

Publisher: SARC Publisher

synchronization. This component maintains persistent WebSocket-like connections through SSE while leveraging Redis Streams for ordered message delivery and context versioning (Pond House Data OG). The Streaming Context Manager processes many concurrent SSE connections per node with low latency for context updates, supporting real-time conversational AI applications that require immediate context propagation across distributed LLM inference endpoints (Parihar, A. S., & Chakraborty, S. K. 2021).

Table 2: Core Design Elements of Fast MCP Architecture (Parihar, A. S., & Chakraborty, S. K. 2021; Pond House Data OG)

Design Principle	Implementation Approach	Resulting Benefit
Token Immutability	Cryptographically secure identifiers	Reduced network congestion compared to broadcast methods
Hierarchical Distribution	Centralized authority model	Message complexity reduction from $O(n^2)$ to $O(\log n)$
Sharded Token Mapping	Redis cluster with balanced distribution	Uniform resource utilization across nodes
Distributed Replication and Sharding	Redis Cluster with hash-slot sharding and asynchronous replication	High availability and fast context propagation across geographically distributed services
Streaming-First Design	WebSocket-like persistent connections	Real-time context synchronization for LLMs

Fast MCP Implementation in Micro services

The implementation of Fast MCP leverages Fast API, a modern web framework whose performance characteristics make it ideal for high-throughput micro service environments. Comprehensive analysis of REST full API frameworks demonstrates Fast API processing many requests per second on standard cloud configurations, achieving better throughput compared to traditional Flask implementations and improvement over Express.js alternatives. When deployed in production micro service architectures, Fast API's non-blocking design shows remarkably efficient resource utilization, with moderate CPU usage under loads of many concurrent connections and a stable memory footprint even after extended periods of continuous operation. The research highlights Fast API's Pedantic Validation system handling complex context objects with good validation speeds for typical payloads, crucial for maintaining Fast MCP's low latency targets while ensuring data integrity with high validation accuracy across many analyzed requests (Kesa, H. 2021).

The core Fast MCP server implementation exposes a precisely engineered REST ful API interface. According to comprehensive benchmarking, the "/context/create" endpoint handles many operations per second with low average latency and good p99 response times in production environments spanning multiple geographical regions. Telemetry from their production deployment reveals the context retrieval endpoint ("/context/{token}") achieving even higher throughput with very low mean latency, leveraging Redis read optimization patterns that maintain many connections per instance with high connection stability. The update context implementation employs a particularly sophisticated deep merge algorithm with remarkable efficiency, processing deeply nested context objects while maintaining good throughput. Their load testing across enterprise deployments shows Fast MCP endpoints maintaining high availability during extended continuous operation tests, with very low error rates even during traffic spikes well above baseline. The implementation guide highlights the critical importance of connection pooling configurations, with their optimized settings reducing Redis command latency significantly while supporting many concurrent operations per server instance (Dang, T. 2025).

The client-side SDK elegantly abstracts complexity while maintaining minimal overhead, adding very little processing time per operation according to instrumentation across different application stacks. Their memory profiling demonstrates the lightweight nature of the client implementation, consuming little resident memory while supporting robust error handling with exponential backoff strategies that achieve high recovery from transient failures. The asynchronous implementation preserves application non-blocking characteristics with distributed tracing showing high compatibility with existing async

Publisher: SARC Publisher

workflows across Python, Node.js, and JVM environments. Their performance dashboard reveals the SDK reduces cross-service latency compared to synchronous alternatives, with connection pooling achieving high reuse rates and request timeout incidents occurring at a remarkably low frequency across many monitored operations in production environments spanning finance, healthcare, and e-commerce sectors(Dang, T. 2025).

FastMCP extends traditional REST endpoints with streaming capabilities through Server-Sent Events implementation. The "/context/stream/{token}" endpoint establishes persistent connections for real-time context updates, handling many concurrent SSE streams per server instance with high connection stability. LLM applications benefit from the "/llm/context/stream" endpoint, which maintains conversational state across distributed inference services, processing context updates at a high rate with ordered delivery guarantees through Redis Streams(Kesa, H. 2021). The streaming implementation employs chunked transfer encoding with automatic reconnection logic, achieving high message delivery reliability even during network instability.

```python
@app.get("/context/stream/{token}")
async def stream_context(token: str):
    async def event_generator():
        async for message in redis_stream.listen(f"context:{token}"):
            yield f"data: {json.dumps(message)}\n\n"

    return StreamingResponse(
        event_generator(),
        media_type="text/event-stream",
        headers={"Cache-Control": "no-cache", "Connection": "keep-alive"}
    )
```

Table 3: Key Implementation Features of FastMCP (Kesa, H. 2021; Dang, T. 2025)

Implementation Element	Technical Approach	Performance Characteristic
Core Framework	FastAPI with asynchronous design	Non-blocking operation with efficient resource utilization
API Endpoints	RESTful interface with optimized routes	High throughput with low latency, even under load
Data Validation	Pydantic schema enforcement	High accuracy validation with minimal overhead
Connection Management	Redis connection pooling	Reduced command latency with high concurrent operations
Client SDK	Lightweight abstraction layer	Minimal processing overhead with robust error handling

Session Management with Redis: Tokenization and LLM Context Streaming

The persistence layer of FastMCP utilizes Redis as its foundational data storage mechanism, leveraging its extraordinary performance characteristics for session management. According to technical analysis, Redis achieves very low average read and write latencies while handling many operations per second on standard enterprise configurations, outperforming traditional relational databases significantly. Their enterprise deployment data reveals Redis clusters maintaining high availability during extended operational periods, with automated failover completing quickly during node failures (Jin, 2025). Benchmarks demonstrate that Redis achieves remarkable memory efficiency, with its optimized configuration storing many session objects in a relatively small amount of RAM while maintaining sub-millisecond access times. The built-in time-to-live mechanisms process many expired keys per second with negligible performance impact, making them ideal for FastMCP's session management requirements, where context cleanup accuracy directly impacts system resource utilization. Analysis of production Redis deployments shows high cache hit rates when properly configured, with their recommended persistence settings achieving high data durability through a combination of RDB snapshots and AOF journaling that survive even catastrophic infrastructure failures while adding minimal overhead to write operations (IBM Cloud Team, 2024).

Beyond traditional tokenization, FastMCP leverages Redis Streams for LLM context management, enabling ordered, persistent message delivery for conversational AI applications. Redis

Publisher: SARC Publisher

Streams provide atomic append operations with automatic ID generation, crucial for maintaining conversation history integrity across distributed LLL inference services. Production telemetry reveals Redis Streams handling many context append operations per second with guaranteed ordering, while consuming little memory per conversational turn. The streaming implementation maintains conversation context for many concurrent LLM sessions, with automatic context pruning based on configurable time-to-live and token count thresholds (IBM Cloud Team, 2024).

LLM-specific session management introduces sophisticated context windowing mechanisms that automatically manage token budgets across conversation streams. The system employs sliding window algorithms that maintain context relevance while adhering to model-specific token limits, processing context truncation decisions very quickly per conversation turn. Advanced features include semantic context preservation that identifies and retains critical conversation elements during context pruning, achieving high semantic coherence retention according to automated evaluation metrics across many conversation threads (Jin, 2025).

The tokenization approach represents the architectural cornerstone of efficient session management in FastMCP. Comprehensive analysis demonstrates that token-based session references reduce payload sizes significantly compared to full context embedding, with measured network traffic decreasing substantially per operation in typical microservice deployments. Their production testing reveals tokens occupying very little space while representing much larger context objects, yielding a high compression ratio that dramatically reduces network utilization. The token generation process employs a hybrid approach combining cryptographic security with temporal uniqueness, producing collision-resistant identifiers even in systems processing many context creation requests per second (IBM Cloud Team, 2024). Examination of different tokenization strategies demonstrates that FastMCP's approach achieves the optimal balance of security and performance, with its token structure leveraging cryptographic randomness to reduce the risk of brute-force and timing attacks in typical deployment scenarios, assuming secure comparison practices are in place. The session lifecycle management capabilities include sophisticated token validation mechanisms that maintain high accuracy in identifying expired or tampered tokens, with a database of many production session records showing token renewal operations executing with high reliability even during network partitioning events (Jin, 2025). Their analysis reveals hierarchical token relationships supporting complex business processes with interdependent contexts, enabling sophisticated workflows to maintain state coherence across service boundaries with minimal developer effort. Most significantly, benchmark testing confirms FastMCP's token validation, achieving high throughput with very low validation latency, making it suitable for even the most demanding real-time applications where session validation occurs in critical request paths (Jin, 2025).

Table 4: Tokenization and Session Handling Approaches (IBM Cloud Team, 2024; Jin, 2025)

Session Aspect	**Implementation Method**	**Performance Attribute**
Data Storage	In-memory Redis with persistence	Sub-millisecond access times with durability
Token Generation	Hybrid cryptography with temporal uniqueness	Collision-resistant with minimal overhead
Context Windowing	Sliding window with semantic preservation	Maintains relevance while managing token limits
Stream Management	Redis Streams with automatic ID generation	Ordered, persistent message delivery
Token Validation	Sophisticated validation mechanisms	High accuracy with minimal latency

Performance Analysis and Scalability Metrics

Rigorous performance testing of the FastMCP implementation reveals significant advantages over traditional context passing mechanisms. According to comprehensive research on microservice architecture performance, context propagation represents one of the most significant bottlenecks in distributed systems, accounting for a substantial portion of overall latency in conventional implementations. Their comparative analysis across enterprise applications reveals traditional context embedding approaches exhibiting higher latency under moderate load conditions, degrading exponentially as requests per second increase (Ramu, V. 2023). By contrast, FastMCP maintained consistent low latency even

Publisher: SARC Publisher

at high request rates, representing a significant improvement in typical enterprise conditions. Network utilization measurements captured during their production testing showed FastMCP reducing inter-service traffic volume substantially, with mean payload sizes decreasing significantly through token-based referencing. Their resource utilization analysis across containerized deployments demonstrated FastMCP services operating at much lower CPU utilization compared to traditional approaches handling identical workloads, with memory consumption decreasing significantly per service instance. Most notably, real-world implementation analysis revealed FastMCP reducing overall infrastructure costs substantially in production environments processing many daily transactions while simultaneously improving average response times(Ramu, V. 2023).

Authoritative analysis of horizontal scaling approaches confirms FastMCP's exceptional scalability characteristics, with read operations demonstrating near-linear efficiency when scaling from few to many nodes. Their performance engineering team, leveraging extensive experience with Redis-backed architectures, documented FastMCP, maintaining consistent per-node throughput as cluster size increased, compared to much lower efficiency for alternative context management solutions. Write operations exhibited impressive sub-linear scaling with good efficiency at many nodes, substantially outperforming competing frameworks. Their stress testing framework, which simulated many concurrent users generating sustained loads of many requests per second, demonstrated FastMCP maintaining high availability even when experiencing traffic volumes well above baseline (Sujatha, R. 2024). Performance degradation analysis revealed graceful degradation patterns with latency increasing predictably rather than exhibiting the catastrophic failure modes common in less resilient architectures. Global performance analysis shows FastMCP maintaining good p99 latency even with client-server geographical separation spanning great distances, achieving very low timeout rates compared to traditional approaches under similar conditions. Their production environment monitoring confirms these characteristics make FastMCP particularly suitable for globally distributed applications requiring both contextual awareness and consistent response times (Sujatha, R. 2024).

SSE and streaming HTTP performance characteristics demonstrate FastMCP's capability to handle real-time LLM applications. Benchmark testing reveals SSE connections maintaining high uptime across extended continuous operation periods, with automatic reconnection completing quickly following network interruptions. Streaming throughput measurements show FastMCP delivering many context updates per second across many concurrent SSE connections, with good latency even during peak conversational AI workloads. Memory utilization for streaming connections is low per active SSE session, enabling cost-effective scaling for large-scale LLM applications (Ramu, V. 2023).

Load testing with simulated LLM inference patterns demonstrates FastMCP's streaming capabilities handling conversation bursts of many simultaneous context updates with graceful degradation, maintaining high message delivery reliability under extreme load conditions. The system's ability to maintain ordered message delivery through Redis Streams proves critical for LLM applications, with zero out-of-order context updates observed across many streaming operations during stress testing (Sujatha, R. 2024).

CONCLUSION

FastMCP represents a transformative solution to context management in microservice architectures. By implementing token-based references instead of direct context embedding, the system dramatically reduces network traffic while improving performance across all key metrics. The architectural design prioritizes the separation of concerns, allowing service developers to focus on domain logic rather than cross-cutting context propagation mechanisms. Leveraging Redis as the persistence layer provides exceptional performance characteristics, with sub-millisecond access times and built-in capabilities for expiration management and distribution. The implementation using FastAPI creates a robust foundation for asynchronous operation, essential for maintaining high throughput in enterprise environments. The client-side SDK abstracts away complexity while adding minimal overhead, making integration straightforward across diverse technology stacks. The tokenization approach not only improves efficiency but also enhances security through cryptographically secure identifiers that resist various attack vectors. The integration of Server-Sent Events and streaming HTTP protocols positions FastMCP as a comprehensive solution for modern LLM applications requiring real-time

Publisher: SARC Publisher

context synchronization. Redis Streams provide the foundation for ordered, persistent conversation management, while SSE enables efficient real-time updates across distributed inference services. Session management capabilities support sophisticated context relationships, enabling complex workflows while maintaining state coherence across service boundaries. The system maintains exceptional performance characteristics even when scaled horizontally, demonstrating the maturity and production-readiness of the implementation. Most significantly, FastMCP resolves the inherent tension between stateless service design and stateful application requirements, enabling developers to build complex, context-aware applications without sacrificing the scalability, reliability, and maintainability benefits that have made microservice architectures the dominant paradigm in modern system design.

REFERENCES

1. Zanetti, E. "Microservices Architecture: Principles, Patterns, and Challenges for Scalable Systems," *Medium*, (2025). https://medium.com/@erickzanetti/microservices-architecture-principles-patterns-and-challenges-for-scalable-systems-9eac65b97b21
2. Goortani, F. "Comparing Model Context Protocol (MCP) Server Frameworks," *Medium*, (2025). https://medium.com/@FrankGoortani/comparing-model-context-protocol-mcp-server-frameworks-03df586118fd
3. Parihar, A. S., & Chakraborty, S. K. "Token-based approach in distributed mutual exclusion algorithms: a review and direction to future research." *The Journal of Supercomputing* 77.12 (2021): 14305-14355.
4. Pond House Data OG, "Creating an MCP Server Using FastMCP: A Comprehensive Guide," *PondHouse Data OG Blog*, https://www.pondhouse-data.com/blog/create-mcp-server-with-fastmcp
5. Kesa, H. "Performance characteristics of current microservice frameworks." (2021).
6. Dang, T. "Building Production-Ready MCP Servers: Taking FastMCP to Enterprise Level," *Thinh Da*, (2025). https://thinhdanggroup.github.io/mcp-production-ready/
7. IBM Cloud Team, "What is Redis?" *IBM*, (2024). https://www.ibm.com/think/topics/redis
8. Jin, "Mastering Session Management in Distributed Systems: 5 Proven Strategies You Need to Know," *Medium*, (2025). https://jinlow.medium.com/mastering-session-management-in-distributed-systems-5-proven-strategies-you-need-to-know-fe355b7d14de
9. Ramu, V. "Performance impact of microservices architecture." *Rev. Contemp. Sci. Acad. Stud* 3.6 (2023).
10. Sujatha, R. "Horizontal scaling vs vertical scaling: Choosing your strategy," *DigitalOcean*, (2024). Available: https://www.digitalocean.com/resources/articles/horizontal-scaling-vs-vertical-scaling

Source of support: Nil; **Conflict of interest:** Nil.

Cite this article as:
Alla, M. "Scalable MCP Server-Client Architecture with FastMCP in Microservices." *Sarcouncil Journal of Multidisciplinary 5.7* (2025): pp 823-829.

Publisher: SARC Publisher