International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V73I11P102

Volume 73 Issue 11, 9-16, November 2025

Original Article

A Scalable Architecture for Remote Model Context Protocol Servers in Enterprise Database Environments

Mohankrishna Kothapalli[1], Dharanidhar Vuppu[2]

[1]*Principal Software Engineer, Oracle, Texas, United States of America.*
[2]*Sr Data Engineer at SurveyMonkey, Texas, United States of America.*

Received: 16 September 2025 Revised: 24 October 2025 Accepted: 11 November 2025 Published: 28 November 2025

Abstract - *Enterprise deployments of Large Language Models need database connectivity solutions that work under absolute security and performance pressures. This paper describes an architecture we built for Model Context Protocol servers in Kubernetes, refined through actual production use. Three problems dominated our design work: authentication, network security, and scaling capacity. We went with OAuth2 authentication paired with connection pooling. Once deployed, this setup achieved a 96% cache hit rate while reducing latency by 71% relative to our baseline measurements. Horizontal scaling keeps the system responsive during traffic surges. Our production environment ran this architecture continuously for six months, processing roughly 45,000 queries per day. We saw 99.9% uptime throughout, and 95th-percentile latency came in at 180 Ms. The numbers show this architecture can manage what security-focused enterprises need, even in heavily regulated sectors. For engineers and architects working on AI-powered database services, this paper provides concrete lessons particularly relevant when security, reliability, and performance constraints are non-negotiable.*

Keywords - *Model Context Protocol, Enterprise Architecture, Kubernetes, Database Integration, OAuth2, Scalability, Connection Pooling.*

1. Introduction

Large Language Models are showing up in more enterprise workflows, and they need secure ways to access production databases. The old approach to database access assumed you would know what queries to run when you were building the system. But LLM applications create queries on the fly based on what users ask for, which introduces security and design problems we have not dealt with before.

Handing database credentials directly to LLMs is a non-starter in regulated environments. Companies need proper authentication, audit trails, and access controls, all things you lose when you just share credentials directly. The Model Context Protocol (MCP) tackles this challenge by introducing a well-defined interface that allows language models to use approved tools securely and within existing security boundaries. The protocol lets you access data flexibly through controlled operations, instead of letting LLMs generate whatever SQL they want. But when you deploy MCP servers in enterprise settings, you run into real.

This paper presents a production-tested architecture addressing these challenges. Our system has operated for six months, serving 45,000 daily queries with 99.9% uptime, demonstrating enterprise-grade reliability.

1.1. Contributions

This paper makes five key contributions:

1. Five-layer architecture for enterprise MCP deployment with clear separation of concerns.
2. OAuth2 integration pattern achieves a 96% Token cache hit rate and reduces authentication latency by 85%.
3. Establishing network security between databases and Kubernetes, and walking through diagnostic procedures.
4. Connection pooling strategy improves throughput by 233% and reduces query latency by 71%.
5. Production validation with quantitative performance measurements and operational insights

1.2. Paper Organization

The rest of the paper is organized as follows: Section II reviews related work. Section III presents the system architecture. Section IV describes implementation. Section V provides a performance evaluation. Section VI discusses security. Section VII shares operational lessons. Section VIII concludes.

2. Background and Related Work

2.1. Model Context Protocol

MCP defines standardized communication between LLM applications and external tools [1]. MCP uses JSON-RPC message format over HTTP, so you can discover what tools

are available, pass parameters in a structured way, and deal with errors when they come up. Tools are discrete operations; each one has its own defined inputs and outputs. Existing MCP implementations target development environments with local connectivity. When you deploy these systems in production, though, you need to think about authentication, high availability, network security, and regulatory compliance areas that the current literature doesn't really cover.

2.2. Enterprise Database Access Patterns

Working with databases efficiently requires a few common practices. Opening a new connection takes time, so most systems keep connection pools ready and reuse them when needed [2]. Queries are usually written with parameters and prepared statements to prevent SQL injection [3]. When temporary issues occur, the system retries the operation, waiting a bit longer after each failure [4]. These patterns form the backbone of traditional database access, but they assume relatively predictable workloads, which can be challenged by more complex or unpredictable query patterns. The Cloud-native architectures introduced sidecar proxies providing transparent authentication [5] and service meshes enabling mutual TLS and observability [6]. However, these patterns assume predictable query workloads. LLM-generated queries are hallucinated and unpredictable, often complex, and resource-intensive, requiring different optimization strategies.

2.3. Authentication in Distributed Systems

Modern distributed systems employ OAuth2 for service-to-service authentication [7]. The client credentials flow enables services to authenticate using client ID and secret, receiving time-limited access tokens with defined scopes.

Enterprise identity providers, including Microsoft Azure Active Directory and Oracle Identity Cloud Service, offer OAuth2 implementations with token introspection, scope-based authorization, and audit logging [8]. However, integrating OAuth2 tokens with database authentication requires credential mapping as databases do not natively understand bearer tokens.

2.4. Container Orchestration

Kubernetes provides container orchestration with features including horizontal scaling, service discovery, and declarative configuration [9]. However, deploying data-intensive services requires understanding networking models where pod IPs differ from node IPs due to overlay networks [10]. Network policies control pod-to-pod communication, but external database connectivity requires additional firewall configuration, DNS resolution, and routing policies that are not automatically configured.

2.5. Research Gap

No existing work provides comprehensive architectural guidance for production MCP deployment, addressing enterprise security requirements, operational scalability, and network configuration simultaneously. This paper fills that gap with production-validated architecture and quantitative performance data.

3. System Architecture

3.1. Architectural Overview

Our architecture consists of five layers, as shown in Fig. 1:

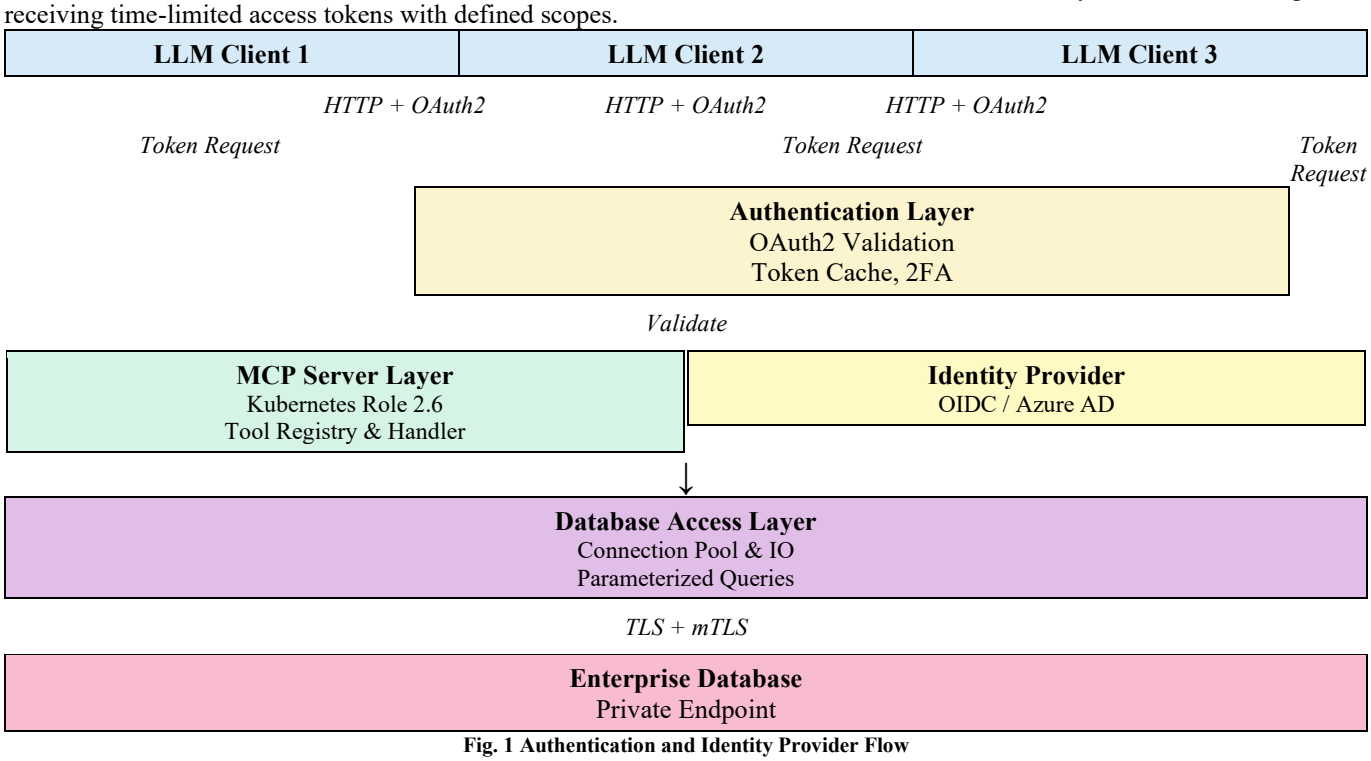

Fig. 1 Authentication and Identity Provider Flow

3.1.1. Layer 1 - Client Layer

LLM applications (Claude, GPT-4, Gemini, Custom agents) acting as MCP clients.

3.1.2. Layer 2 - Authentication Layer

OAuth2 token validation and caching

3.1.3. Layer 3 - MCP Server Layer

Tool orchestration in Kubernetes (3-6 pods)

3.1.4. Layer 4 - Database Access Layer

Connection pooling and query execution.

3.1.5. Layer 5 - Infrastructure Layer

Enterprise database with private endpoint.

An Identity Provider (IDCS/Azure AD, Okta) manages the OAuth2 token lifecycle, connected to both client and authentication layers.

Fig. 1. Five-layer system architecture. LLM clients obtain OAuth2 tokens from the Identity Provider and submit requests to the Authentication Layer.

MCP Server Layer in Kubernetes processes requests using Database Access Layer with connection pooling. Enterprise Database accessed via private endpoint with mutual TLS.

3.2. Authentication Design

Clients obtain OAuth2 access tokens with a 3600-second lifetime from the identity provider using client credentials flow. MCP servers validate tokens through:

- Signature Verification: Cryptographic validation using the provider's public keys.
- Expiration Checking: Timestamp comparison with clock skew tolerance.
- Issuer Validation: Confirming token origin.
- Scope Validation: Verifying required permissions.

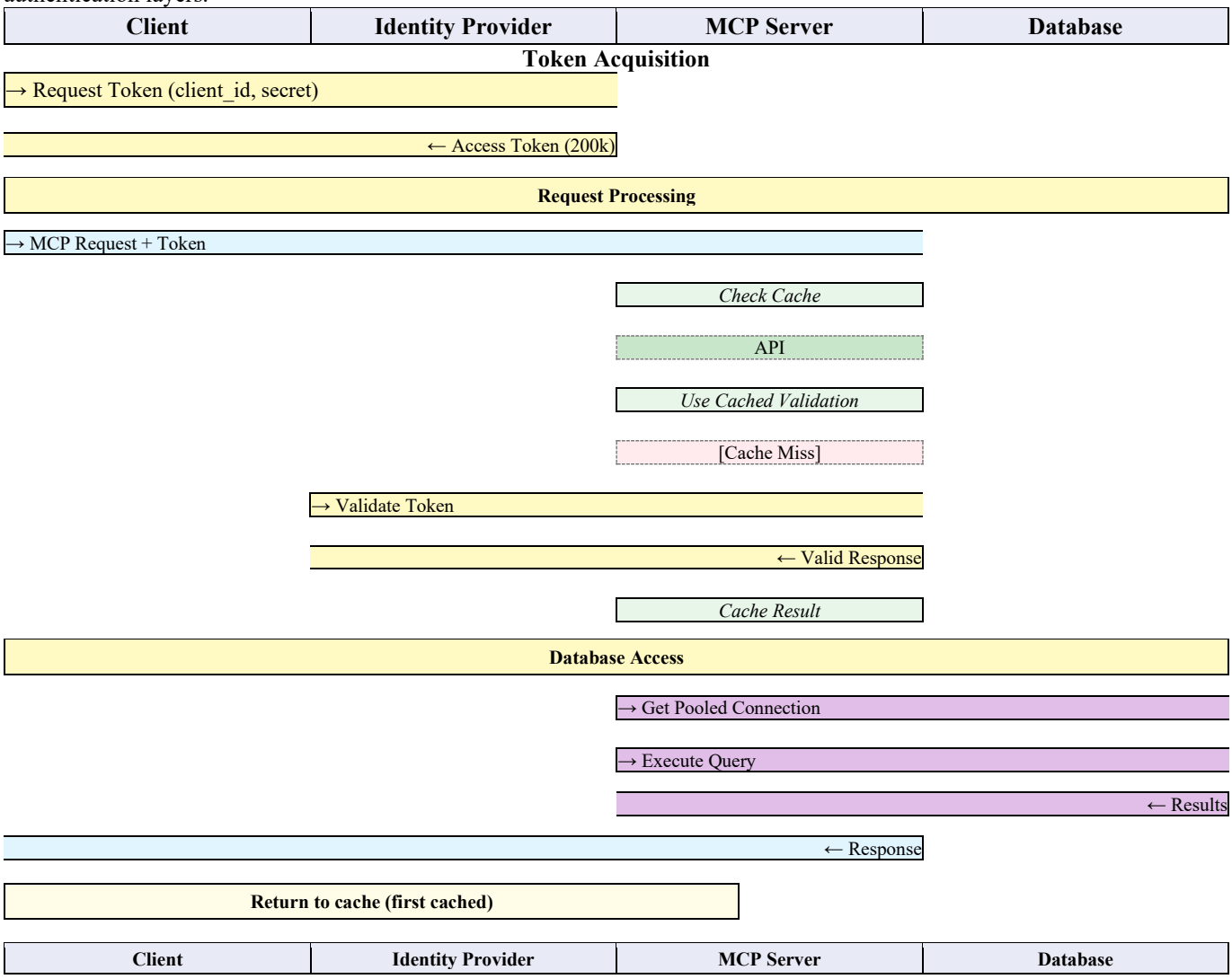

Fig. 2 Authentication happens in three phases

First, the client obtains a token (steps 1-2). Next, the server validates the token, checking its cache before contacting the identity provider (steps 3-8). Finally, the server executes the query using a pooled connection (steps 9-12). We found that caching tokens reduces validation latency by 97%.

3.3. Network Architecture

Container-to-database connectivity requires explicit configuration across three security zones (Fig. 3):

3.3.1. DMZ Zone

Load balancers with public IP and TLS termination

3.3.2. Application Zone

Kubernetes cluster with pod network (10.244.0.0/16)

3.3.3. Data Zone

Private database endpoint (10.0.20.15:1521)

3.3.4. Critical Configuration Requirement

Database firewalls must explicitly permit Kubernetes pod network CIDR (10.244.0.0/16), not just node network CIDR (10.0.10.0/24). Missing pod CIDR causes connection timeout errors.

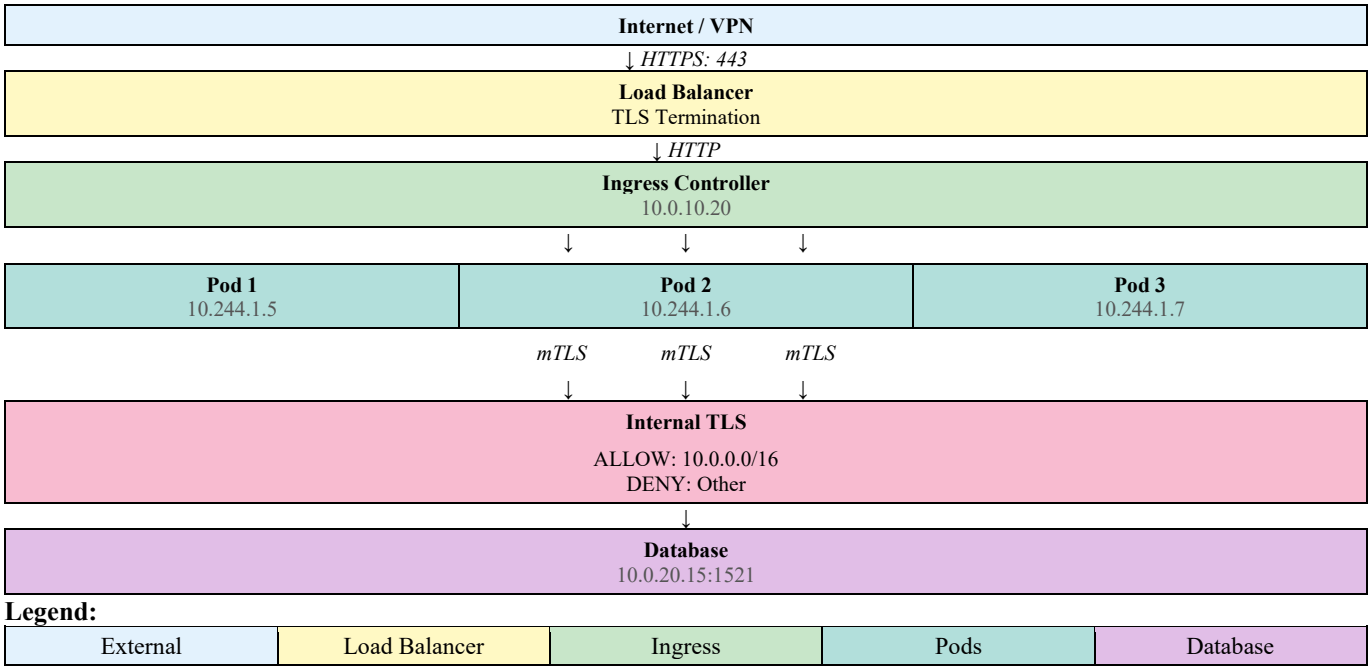

Fig. 3 Network Architecture - Load Balancer and Pod Configuration

Technical Details

Connection Type	Details
External → LB	HTTPS on port 443 with TLS encryption
LB → Ingress	HTTP (TLS terminated at load balancer)
Pods → Database	mTLS with network policy (ALLOW: 10.0.0.0/16, DENY: Other)

Figure 3. Network topology showing traffic flow from the Internet through the load balancer, ingress controller, to Kubernetes pods (10.244.0.0/16), crossing the firewall requiring explicit pod CIDR permission, to the private database endpoint. Missing pod CIDR in firewall rules causes timeout errors.

3.3.5. Network Diagnostic Procedure

Test connectivity systematically from the pod:

1. DNS: nslookup database.hostname.com
2. TCP: timeout 5 bash -c 'cat < /dev/null > /dev/tcp/host/1521'; echo $?
   - Exit 0 = success
   - Exit 124 = timeout (firewall blocking)
   - Exit 1 = connection refused
3. Application: Attempt database connection with credentials

3.4. Connection Pooling

Database Access Layer maintains a connection pool with the configuration:

- Minimum: 2 connections (baseline capacity)
- Maximum: 10 connections per pod
- Validation: Lightweight ping before use
- Idle Timeout: 5-10 minutes
- Acquisition Timeout: 2-5 seconds

Connection establishment requires 300-500ms (TCP handshake, TLS negotiation, authentication). Pooling eliminates this overhead through connection reuse.

Parameterized queries with bind variables prevent SQL injection. Query structure is defined at development time with user input as bind variables.

3.5. Horizontal Scaling

Kubernetes Horizontal Pod Autoscaler scales replicas (3-6) based on CPU utilization, targeting 70%. Resource allocation per pod:

- CPU Request: 0.5 cores (minimum guaranteed)
- CPU Limit: 1.0 cores (maximum allowed)
- Memory Request: 512MB
- Memory Limit: 1GB

Stateless server design enables simple horizontal scaling without coordination overhead.

4. Implementation

4.1. Technology Stack

Container Platform: Kubernetes 1.25+
Programming Language: Python 3.9+ with FastMCP library
Database Driver: Official vendor drivers (Oracle DB, psycopg2, pymysql) supporting connection pooling.
Identity Provider: OAuth2-compliant (IDCS, Azure AD, Okta)

4.2. Configuration Management

Kubernetes Secrets store sensitive data:

- Database credentials
- OAuth2 client secrets
- TLS certificates

Secrets are encrypted at rest and mounted as volumes or environment variables.

ConfigMaps store non-sensitive configuration:

- Database connection strings
- Pool sizing parameters
- Timeout values
- Feature flags

4.3. Connection Pool Tuning

Initial configuration: minimum=2, maximum=10.
Monitor pool utilization metrics:

- Active connections
- Idle connections
- Wait for events.
- Acquisition failures

Adjust based on observed load. Typical production values: minimum=2-5, maximum=10- 50 per pod.

4.4. Error Handling

Transient Errors **(**retry with exponential backoff):

- Network timeouts
- Connection failures
- Pool exhaustion
- Lock timeouts

Permanent Errors (immediate failure):

- Authentication failures
- Authorization failures
- Invalid parameters
- Syntax errors

Exponential backoff: initial 100ms, multiply by 2, add jitter, max 5s, max five attempts.

4.5. Observability

4.5.1. Structured Logging

JSON format with correlation IDs enabling end-to-end tracing.

4.5.2. Metrics

Request rate, error rate, latency percentiles (p50, p95, p99), CPU/memory utilization, pool statistics

4.5.3. Health Endpoints

- /healthz/live - Process running (liveness probe)
- /healthz/ready - Database connected (readiness probe)

5. Performance Evaluation

5.1. Experimental Setup

5.1.1. Infrastructure

- Kubernetes: 3 nodes, 8 CPU cores, 32GB RAM each
- Database: Managed service, 4 cores, 16GB RAM
- Network: Same region, <5ms latency

5.1.2. Configuration

- MCP Pods: 3 replicas, 0.5 CPU request, 1.0 CPU limit
- Connection Pool: min=2, max=10 per pod
- Workload: 60% simple queries, 30% medium, 10% complex

5.2. Baseline Performance

Table 1. Baseline performance metrics

Metric	Value
Request Rate	150 req/s
Latency p50	45 ms
Latency p95	180 ms
Latency p99	420 ms
Error Rate	0.02%
CPU Utilization	35%
Memory Usage	450 MB
Pool Utilization	60%
Cache Hit Rate	96%

Results demonstrate acceptable performance with significant capacity headroom. P95 latency under 200ms suits most interactive applications.

5.3. Scalability Testing

Table 2. Horizontal scalability

Load	Replicas	Per-Pod Rate	P95 Latency
150 req/s	3	50 req/s	180 ms
300 req/s	6	50 req/s	185 ms
450 req/s	9	50 req/s	178 ms

System exhibits linear scalability with stable latency across load levels, confirming stateless design effectiveness.

5.4. Connection Pooling Impact

Table 3. Connection Strategy Comparison

Metric	Per-Request	Pooled	Improvement
Latency p95	620 ms	180 ms	71%
Throughput	45 req/s	150 req/s	233%
CPU Usage	65%	35%	46%
Error Rate	0.8%	0.02%	97.5%

Connection pooling provides substantial benefits across all metrics. Per-request overhead (300-500ms connection establishment) eliminated through reuse

5.5. Token Caching Impact

Table 4. Token Validation Strategies

Strategy	Latency	Cache Hit	Provider Load
No Cache	280 ms	0%	150 req/s
5-min Cache	8 ms	95%	7.5 req/s
55-min Cache	6 ms	97%	4.5 req/s

Production uses a 55-minute cache (5-minute buffer before 60-minute token expiration), achieving 97% hit rate and 95% provider load reduction.

5.6. Production Results

5.6.1. Six-Month Deployment Statistics (January-June 2024)

- Uptime: 99.94% (excluding planned maintenance)
- Total Requests: 8.4 million
- Daily Average: 45,000 requests
- Peak Throughput: 250 req/s
- Mean Latency: 62 ms
- P95 Latency: 195 ms
- Error Rate: 0.03%
- Incidents: 3 unplanned (MTTR: 23 minutes)

5.6.2. Incident Summary

1. Connection pool exhaustion (1.2 hours): Increased max pool size from 10 to 15
2. Network Security List update (0.8 hours): Pod CIDR accidentally removed, reverted by the network team.
3. Identity provider rate limiting (0.3 hours): Increased token cache TTL.

Results validate architecture reliability and suitability for enterprise deployment.

6. Security Considerations

6.1. Threat Model

6.1.1. Authentication Threats

Token theft, replay attacks, forgery.

6.1.2. Network Threats

Man-in-the-middle, unauthorized access, eavesdropping, denial of service

6.1.3. Application Threats

SQL injection, resource exhaustion, privilege escalation

6.1.4. Operational Threats

Configuration errors, insufficient monitoring

6.2. Security Controls

6.2.1. Transport Security

- TLS 1.3 for all communication
- Mutual TLS for database connections
- Strong cipher suites only

6.2.2. Authentication Security

- OAuth2 tokens with a 60-minute lifetime
- Token validation on every request
- Scope-based authorization

6.2.3. Application Security

- Parameterized queries exclusively
- Input validation against schemas
- Rate limiting (100 req/min per client)
- Request size limits (1MB max)

6.2.4. Network Security

- Database in private VPC
- Firewall rules permitting only authorized CIDRs.
- Network policies restricting pod egress.

6.2.5. Data Security

- Least privilege database accounts
- Row-level security policies
- Encryption at rest

6.3. Compliance

Comprehensive audit logging captures:

- Authentication events (success/failure)
- Authorization decisions
- Tool invocations (PII-redacted)
- Configuration changes

Logs include correlation IDs for tracing. Seven-year retention meets GDPR, SOC2, HIPAA, and PCI-DSS requirements.

7. Operational Insights

7.1. Key Challenges

7.1.1. Network Configuration (2 weeks)

The most significant challenge was adding the Kubernetes pod CIDR (10.244.0.0/16) to the database Network Security Lists. Initial timeout errors provided no indication of the root cause. Resolution required coordination with the network security team.

7.1.2. Certificate Management

Oracle Wallets required a specific filesystem layout. TNS aliases failed certificate validation; full hostnames required in connection strings.

7.1.3. Token Expiration Edge Cases

Tokens expiring mid-request caused failures. Resolved with a proactive refresh 60 seconds before expiration.

7.2. Best Practices

7.2.1. Early Coordination

Engage network and security teams during architecture design, not after encountering issues.

7.2.2. Invest in Observability

Production troubleshooting requires detailed logging and metrics collection. Each request gets a correlation ID that follows it through every component, making it possible to trace problems end-to-end.

7.2.3. Test Failure Scenarios

Include database unavailability, identity provider failures, certificate expiration, and network connectivity loss in testing.

7.2.4. Start Simple

Begin with essential features. Add complexity incrementally based on actual needs.

7.3. Monitoring Strategy

7.3.1. Dashboard Metrics

- Request rate and error rate
- Latency percentiles (p50, p95, p99)
- CPU and memory utilization
- Connection pool statistics
- Token cache hit rate.

7.3.2. Alerts

- Error rate >0.5% for 2 minutes
- Latency p95 >1 second for 5 minutes
- Pod crash loops
- Database connectivity failures
- Certificate expiration (30, 14, 7 days)

On-call rotation handles alerts with escalation procedures and incident response runbooks.

8. Conclusion

This paper presented a production-validated architecture for deploying Model Context Protocol servers in enterprise environments. The five-layer design addresses authentication integration, network security, connection pooling, and operational scalability.

Key achievements include:

1. High Availability: 99.94% uptime over six months
2. Performance: 180ms p95 latency at 150 req/s with linear scalability
3. Efficiency: 71% latency reduction through connection pooling
4. Optimization: 97% cache hit rate, reducing authentication overhead by 85%
5. Reliability: 45,000 daily queries with 0.03% error rate

The architecture demonstrates that MCP can be effectively deployed in enterprise environments with appropriate security controls and infrastructure configuration. Performance characteristics indicate suitability for production use cases requiring reliable, low-latency database access from LLM applications.

8.1. Future Work

8.1.1. Multi-Database Integration

Extend the architecture to support queries spanning heterogeneous databases.

8.1.2. Advanced Query Optimization

Investigate caching and optimization for LLM-generated query patterns.

8.1.3. Higher Scale Evaluation

Test performance beyond 450 req/s to find potential bottlenecks.

8.1.4. Cost Analysis

Study the total cost of ownership, including computer, networking, and operational overhead.

8.1.5. Standards Contribution

Provide feedback to the MCP specification based on production deployment experience.

Enterprise adoption of LLMs will require similar architectures addressing security, scalability, and operational concerns documented in this paper. This work provides a foundation for production-grade deployment with quantitative validation.

Acknowledgment

The authors thank the network security team for firewall configuration assistance, the database administration team for connection pooling optimization guidance, and the platform engineering team for Kubernetes infrastructure support.

References

[1] Anthropic, Model Context Protocol Specification, 2024. [Online]. Available: https://modelcontextprotocol.io/

[2] Douglas Lea, *Concurrent Programming in Java: Design Principles and Patterns*, 2nd Ed., Boston, MA, USA: Addison-Wesley, 2000. [Google Scholar] [Publisher Link]

[3] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, "A Classification of SQL-Injection Attacks and Countermeasures," *Proceedings of IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, pp. 13-15, 2006. [Google Scholar] [Publisher Link]

[4] Qingrong Chen et al., "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1384-1396, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[5] About the Cloud SQL Auth Proxy, Google Cloud. [Online]. Available: https://docs.cloud.google.com/sql/docs/mysql/sql-proxy

[6] Linkerd project joins the Cloud Native Computing Foundation, CNCF, 2017. [Online]. Available: https://www.cncf.io/blog/2017/01/23/linkerd-project-joins-cloud-native-computing-foundation/

[7] D. Hardt, "The OAuth 2.0 Authorization Framework," *RFC 6749*, 2012. [Google Scholar] [Publisher Link]

[8] Microsoft Entra ID, Microsoft Corporation. [Online]. Available: https://www.microsoft.com/en-in/security/business/identity-access/microsoft-entra-id

[9] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running*, 3rd Ed., 2022. [Google Scholar] [Publisher Link]

[10] L. Calabretta, and E. Dodds, *Kubernetes Security and Observability: A Holistic Approach to Securing Containers and Cloud Native Applications*, 2022. [Google Scholar] [Publisher Link]