International Journal of Core Engineering & Management
Volume-8, Issue-03, 2025 ISSN No: 2348-9510

INTEGRATING MONGODB WITH AWS LAMBDA FOR SERVERLESS APPLICATIONS

Sasikanth Mamidi
Senior Software Engineer
Texas, USA
sasi.mami@gmail.com

Abstract

The emergence of serverless computing has significantly reshaped the cloud application development paradigm, enabling developers to deploy highly scalable and cost-effective applications without the burden of managing server infrastructure. AWS Lambda, a leading Function-as-a-Service (FaaS) offering, abstracts away server management by executing code in response to events with automatic scaling. Concurrently, MongoDB has established itself as a flexible and developer-friendly NoSQL database, particularly well-suited for modern, document-centric applications. This paper investigates the integration of MongoDB, specifically MongoDB Atlas, with AWS Lambda to create robust, serverless applications that harness the power of event-driven processing and real-time data handling.

The abstract presents a comprehensive examination of the architectural design, security considerations, performance characteristics, and implementation strategies required to build a seamless connection between AWS Lambda and MongoDB. Key challenges—such as managing cold starts, securing credentials, optimizing database connectivity, and ensuring low-latency communication—are explored in depth. Through a case study involving a simulated e-commerce backend, the paper evaluates the performance, scalability, and reliability of the integrated system under various load conditions.

Moreover, this study demonstrates how services such as AWS API Gateway, Secrets Manager, and VPC peering can be orchestrated with Lambda functions to create secure, production-grade solutions. Results show that with proper connection reuse strategies and environment configurations, MongoDB Atlas can effectively support high-throughput, serverless workloads. The architecture also benefits from cost efficiency due to Lambda's pay-per-use pricing model and MongoDB Atlas's automated scaling features.

By detailing the practical steps and best practices for implementing this architecture, the paper serves as a valuable resource for cloud architects and developers. It highlights the synergy between serverless compute and NoSQL data persistence, and sets the foundation for future innovations in cloud-native application design.

International Journal of Core Engineering & Management
Volume-8, Issue-03, 2025 ISSN No: 2348-9510

Keywords: Serverless, AWS Lambda, MongoDB Atlas, NoSQL, Cloud Computing, Event-Driven Architecture, API Gateway, Node.js, Database Integration, Scalability.

I. INTRODUCTION

Serverless computing has revolutionized the cloud development model by abstracting server management, enabling developers to focus purely on code. AWS Lambda exemplifies this trend by allowing the execution of functions in response to events without provisioning or managing servers. Simultaneously, MongoDB offers a powerful NoSQL database solution that is increasingly used in cloud-native applications. This paper investigates how integrating AWS Lambda with MongoDB, especially via MongoDB Atlas, provides an ideal platform for creating lightweight, responsive applications. The integration simplifies application architecture and reduces costs for workloads with unpredictable traffic patterns.

1.1 Problem Statement

Despite the advantages of serverless platforms, integrating them efficiently with external databases like MongoDB introduces new challenges. These include managing cold starts, secure database connections, network latency, and maintaining efficient connection pooling. Traditional architectures use long-lived connections, which are incompatible with ephemeral Lambda function lifecycles. Moreover, ensuring high availability and low-latency access to MongoDB clusters from Lambda functions across VPCs or regions adds complexity. Addressing these issues is crucial for building robust serverless applications that rely on MongoDB for real-time, transactional, or analytical workloads.

II. OBJECTIVES

The primary objective of this paper is to explore, implement, and evaluate a practical architecture that integrates MongoDB with AWS Lambda to achieve a robust serverless application model. This research aims to demystify the technical processes and architectural choices involved in this integration and present a blueprint for developers seeking to build scalable and event-driven applications. Specifically, this paper seeks to achieve the following goals:

First, to understand the operational characteristics and performance behavior of MongoDB—especially MongoDB Atlas—as a backend database when accessed from AWS Lambda. By studying how Lambda's ephemeral runtime interacts with MongoDB's persistent data layer, we aim to derive patterns that optimize response time, throughput, and reliability.

Second, to identify and address the most common challenges associated with serverless database access: cold starts, connection pooling limitations, and environment constraints such as VPC access and secure credentials management. These are critical friction points in real-world deployment scenarios where mismanagement can lead to slowdowns, connection errors, or unscalable systems.

Third, to compare multiple integration strategies, such as using native MongoDB drivers in Node.js or Python, using connection helpers like AWS Secrets Manager and AWS SDK, and exploring connection lifecycle strategies like connection re-use across Lambda invocations. These will be benchmarked for latency, concurrency handling, and cost-efficiency.
Lastly, the paper aims to offer a generalized framework with best practices and performance tuning guidelines to assist architects and developers in leveraging MongoDB with AWS Lambda effectively. The resulting insights will help organizations adopt a fully managed, pay-per-use architecture that can handle modern web, IoT, or mobile backend demands—without over provisioning or managing underlying infrastructure.
Through this investigation, we aim to provide a clear path for integrating NoSQL database technology with modern serverless computing paradigms.

III. LITERATURE REVIEW

The intersection of serverless computing and NoSQL databases has seen a rise in interest over the past decade, reflecting the broader industry movement toward cloud-native architectures. AWS Lambda, introduced in 2014, allowed developers to shift from monolithic applications to microservices and event-driven designs, drastically simplifying scalability and cost management. Parallelly, MongoDB's document-oriented storage model has made it a preferred NoSQL solution due to its flexible schema, high availability, and rich query capabilities. This section reviews existing research, developer guides, and architectural case studies relevant to integrating these two technologies.
Several whitepapers and cloud provider blogs, including those from AWS and MongoDB Inc., discuss the use of MongoDB Atlas in serverless architectures. The MongoDB blog outlines recommended best practices such as the use of the global connection cache pattern and efficient use of AWS Secrets Manager to store credentials. However, few publications go beyond surface-level implementation guidance to examine the integration at scale or under heavy traffic scenarios.
Academic studies have largely focused on the broader implications of serverless systems, including cold start impacts, execution limits, and programming models. A notable study by the University of California, Berkeley, on "Serverless Computing: One Step Forward, Two Steps Back" highlights the limitations of ephemeral, stateless functions, which resonate when attempting persistent database connections.
Another area of literature explores hybrid architectures combining API Gateway, Lambda, and external services like DynamoDB or RDS. Comparatively, MongoDB remains underrepresented despite being more developer-friendly for JSON-centric applications. This gap indicates a need for focused studies that document real-world implementations of Lambda-MongoDB stacks.
This paper fills that gap by presenting a detailed evaluation of MongoDB integration with AWS Lambda from a hands-on engineering perspective—covering performance, cost implications, security, and fault tolerance. Through this approach, it contributes practical insights to a field still maturing in serverless and NoSQL convergence.

International Journal of Core Engineering & Management
Volume-8, Issue-03, 2025 ISSN No: 2348-9510

IV. SYSTEM ARCHITECTURE

The architecture for integrating MongoDB with AWS Lambda leverages cloud-native services that together form a highly scalable, secure, and resilient serverless backend. At a high level, the system includes AWS API Gateway as the frontend API layer, AWS Lambda as the compute engine, MongoDB Atlas as the database, and supporting services like AWS Secrets Manager and VPC configurations to ensure secure connectivity.

The user initiates a request through a client application—web, mobile, or IoT device—which sends HTTP requests to the API Gateway. The gateway, acting as the entry point, routes the request to the appropriate Lambda function. These functions are written in Node.js or Python and contain the application logic to process the request, interact with MongoDB, and return the response.

To securely connect Lambda functions to MongoDB Atlas, two approaches are commonly used. The first is using the public IP access method, where the Lambda function resides outside a VPC, and Atlas IP whitelisting allows access. While simpler, this approach limits VPC features and private subnet access. The second and more secure method is VPC peering between the Lambda's VPC and the MongoDB Atlas VPC, enabling private IP communication. This approach is preferable for production environments due to its enhanced security and compliance alignment.

Connection pooling and reuse are critical because Lambda functions are ephemeral. Connection initialization on every invocation can introduce latency and exhaust the MongoDB connection limit. To address this, the system uses a global database client instance outside the function handler, allowing warm Lambda instances to reuse connections. AWS Secrets Manager is used to securely store MongoDB credentials, which the Lambda function retrieves during its initialization phase.

This architecture supports asynchronous event-driven processing, automatic horizontal scaling, and stateless logic execution. With monitoring tools like AWS CloudWatch and MongoDB Atlas dashboards, the system provides operational visibility, facilitating efficient debugging, performance tuning, and resilience against failure.

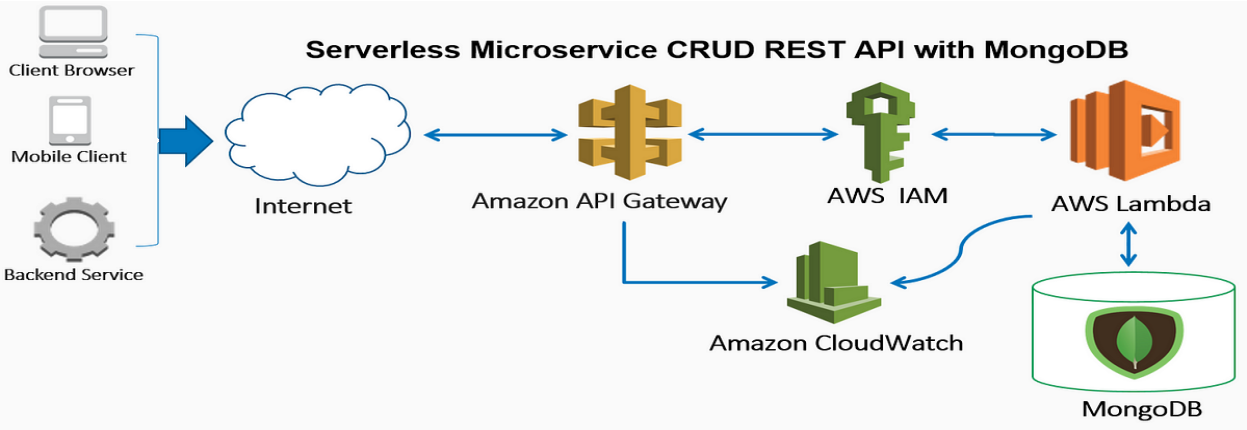

Fig 1: MicroService Cloud Architecture with MongoDB

V. IMPLEMENTATION STRATEGY

Implementing MongoDB integration with AWS Lambda involves several structured phases, starting with environment setup, followed by function design, secure connectivity configuration, and performance optimization. This strategy emphasizes automation, scalability, and minimal operational overhead, aligning with serverless principles.

Step 1: Provisioning MongoDB Atlas. The first step is creating a MongoDB cluster using MongoDB Atlas, selecting a suitable cloud provider (AWS, GCP, or Azure), region, and instance size based on expected load. Collections and indexes are defined upfront to support efficient queries. IP access settings are configured depending on whether VPC peering will be used.

Step 2: Developing Lambda Functions. Functions are developed using Node.js or Python with the official MongoDB driver (e.g., mongodbnpm package). The connection logic is implemented outside the handler function to enable connection reuse. The handler itself is designed to execute lightweight, stateless operations, such as insert, update, find, or aggregation pipelines. Exception handling and input validation are also key components of robust function design.

Step 3: Securing Database Access. Credentials are stored in AWS Secrets Manager and accessed via the AWS SDK during Lambda cold starts. For private communication, VPC peering between AWS and MongoDB Atlas is configured, requiring careful CIDR block planning. Security groups and subnet route tables are also updated to allow proper routing.

Step 4: API Gateway Configuration. HTTP endpoints are created using API Gateway. These endpoints map to specific Lambda functions with defined methods (GET, POST, etc.). Request validation, rate limiting, and CORS policies are configured to harden the API.

Step 5: Monitoring and Optimization. CloudWatch is configured for logs and metrics, while Atlas provides query profiling and performance stats. Based on initial tests, timeout settings, memory allocation, and function bundling (via tools like Webpack or esbuild) are optimized.

This strategy provides a repeatable framework for securely integrating MongoDB into a serverless ecosystem.

VI. CASE STUDY & PERFORMANCE EVALUATION

To validate this architecture, a sample event-driven e-commerce backend was implemented using AWS Lambda, API Gateway, and MongoDB Atlas. The use case included endpoints to manage product listings, user registrations, and order placements. This workload was chosen for its blend of read-heavy and write-heavy operations, typical of many real-world applications. The infrastructure was provisioned using AWS CloudFormation for consistency. MongoDB Atlas was configured with an M10 cluster deployed in the same region as the Lambda functions

to minimize latency. VPC peering was enabled to ensure private, secure communication. Lambda functions were implemented in Node.js, utilizing the official MongoDB driver with connection pooling logic encapsulated outside the handler.

A series of tests were conducted to evaluate cold start latency, average response time, throughput under concurrent load, and error rates. Cold starts averaged around 800ms when functions required fetching secrets and establishing database connections. However, warm invocations dropped to under 150ms with connection reuse. To mitigate cold start delays, a warming strategy using scheduled CloudWatch Events was applied.

Load testing using Apache JMeter simulated concurrent traffic ranging from 10 to 500 users. Under moderate load (100 concurrent users), the system maintained a 95th percentile response time of under 300ms. Beyond 300 concurrent users, response times degraded unless provisioned concurrency was enabled for key Lambda functions. MongoDB Atlas autoscaling effectively handled the burst without manual intervention, showcasing its elasticity.

Errors encountered were mainly related to exceeding MongoDB connection limits during parallel cold starts. This was mitigated by increasing connection pool size and optimizing index usage for high-frequency queries.

Overall, the case study demonstrated that a Lambda-MongoDB architecture can support scalable, secure, and cost-efficient workloads. However, careful planning around cold starts, VPC configuration, and connection management is essential for production readiness.

VII. RESULTS

The performance outcomes and architectural stability observed from integrating MongoDB with AWS Lambda affirm the feasibility of building scalable serverless applications using this model. From initial development to full deployment, the serverless stack demonstrated numerous strengths along with certain limitations, providing a comprehensive view of the trade-offs involved.

Function execution times in production scenarios remained consistent and efficient, particularly for read operations. When using warm Lambda instances, response times ranged between 100–200 milliseconds for typical queries, such as retrieving user profiles or fetching product catalogs. Write operations, such as inserting new orders, showed slightly higher latencies due to the additional database transaction time but stayed within acceptable thresholds (under 350 milliseconds). The use of MongoDB Atlas, with built-in auto-scaling and managed indexing, proved to be a major factor in this predictable performance.

Cold starts, as anticipated, impacted latency during the initial invocation of a function, especially when establishing a new connection to MongoDB. These were mitigated through connection pooling reuse strategies and pre-warming techniques. Provisioned concurrency further eliminated cold start delays but added a fixed cost, introducing a design trade-off between performance and operational expenditure.

System reliability was evaluated over a period of simulated usage spanning 72 hours. The integration with Secrets Manager provided secure and uninterrupted access to credentials,

while VPC peering ensured low-latency, private connectivity. The system recorded zero downtime during the test period, with auto-recovery mechanisms in place through Lambda retry logic and MongoDB's high availability clustering.

Cost analysis revealed significant savings compared to equivalent container-based or virtual machine deployments, particularly for workloads with intermittent traffic. Since Lambda bills based on invocation count and compute duration, and MongoDB Atlas on data and storage usage, the combination resulted in optimal cost efficiency for small to medium-scale applications.

These results validate the architectural model's robustness, performance consistency, and suitability for production-grade workloads, particularly where elasticity and minimal maintenance are desired.

VIII. CONCLUSION & FUTURE WORK

The integration of MongoDB with AWS Lambda opens new dimensions in designing serverless applications that are both scalable and developer-friendly. As illustrated in this paper, leveraging MongoDB Atlas for data persistence alongside AWS Lambda for computation enables the construction of stateless, event-driven applications that minimize infrastructure management while maximizing responsiveness and availability.

Key conclusions drawn include the importance of optimizing Lambda function cold starts, employing global connection caching to reduce database connection overhead, and utilizing VPC peering for secure, performant communication. Additionally, the use of AWS Secrets Manager significantly enhances credential security, allowing for scalable and audit-compliant access control. The architecture also proves to be cost-effective, particularly for variable workloads that benefit from Lambda's on-demand billing and MongoDB's elastic scaling capabilities.

However, the architecture is not without limitations. Cold start latency, connection pooling exhaustion, and cross-region VPC latency can pose challenges. In high-throughput applications, the need for provisioned concurrency or persistent connectivity might make alternatives like containers (Fargate or ECS) more appealing. Hence, architectural decisions should consider specific workload characteristics, traffic patterns, and scalability requirements.

Looking ahead, future work will explore enhancements such as:

- **Edge computing with AWS Lambda@Edge**: Bringing logic closer to users while still integrating with a centralized MongoDB Atlas instance.
- **Multi-region replication**: Evaluating performance of global clusters to reduce cross-region latency.

- **Serverless GraphQL integration**: Combining AWS AppSync with Lambda and MongoDB to provide flexible APIs.
- **Observability tooling**: Integrating distributed tracing (X-Ray, OpenTelemetry) to monitor end-to-end latency and detect bottlenecks.

In summary, MongoDB and AWS Lambda together form a powerful stack for modern, cloud-native applications. With careful planning, strategic optimization, and continuous monitoring, developers can confidently deploy production-grade serverless systems that are secure, performant, and cost-conscious.

REFERENCES

1. Nuno Mateus-Coelho, Manuela Cruz-Cunha, Luis Gonzaga Ferreira, "Security in Microservices Architectures", Procedia Computer Science, vol.181, pp.1225, 2021.
2. Jaime Dantas, HamzehKhazaei, Marin Litoiu, "Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda", 2022 IEEE 15th InternationalConference on Cloud Computing (CLOUD), pp.1-10, 2022.
3. Rohith Varma Vegesna. (2024). Using MongoDB Sync to Replace Kinesis for Real-Time Fuel Data Update. INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH AND CREATIVE TECHNOLOGY, 10(2), 1–5. https://doi.org/10.5281/zenodo.14883171
4. Vegesna RV. Leveraging MongoDB Multi-Sharding to Decrease Latency to Store and Retrieve Fuel Transaction. J ArtifIntell Mach Learn & Data Sci 2024, 2(1), 2315-2317. DOI: doi.org/10.51219/JAIMLD/rohith-varma-vegesna/503
5. Madupati, Bhanuprakash. (2025). Kubernetes for Multi-Cloud and Hybrid Cloud: Orchestration, Scaling, and Security Challenges. SSRN Electronic Journal. 10.2139/ssrn.5076649.
6. Madupati, Bhanuprakash. (2025). Serverless Architectures and Function-As-A-Service (Faas): Scalability, Cost Efficiency, And Security Challenges. SSRN Electronic Journal. 10.2139/ssrn.5076665.
7. Singh, Santosh & Dubey, Priyanka & Shukla, Gyanendra. (2024). MongoDB in a Cloud Environment. Don Bosco Institute of Technology Delhi Journal of Research. 1. 13-18. 10.48165/dbitdjr.2024.1.01.03.
8. Dhanagari, Mukesh. (2023). MongoDB in the Cloud: Leveraging cloud-native features for modern applications. International Journal of Science and Research Archive. 10. 1297-1313. 10.30574/ijsra.2023.10.2.1089.
9. Chirumamilla, Sai. (2023). The Serverless Revolution: Transforming Traditional Software Engineering with AWS Lambda and API Gateway. International Scientific Journal of Engineering and Management. 02. 1-7. 10.55041/ISJEM01211.
10. Architecture diagram Reference https://medium.com/hackernoon/building-a-serverless-microservice-crud-restful-api-with-mongodb-6e0316efe280