Assessing the Effectiveness of Unit and Fuzz Testing in Evaluating MongoDB Reliability in Chameleon Cloud

Youngjae Moon, Lisa Liu, Robert Sheng

CS 4287/5287 Principles of Cloud Computing Final Project

This project explores the use of unit testing and fuzz testing to enhance the deployment and validation of MongoDB clusters in cloud environments using Docker and Kubernetes. MongoDB's growing popularity for its flexibility and scalability underscores the importance of reliable deployment and operational testing in production settings. To address these challenges, we present an automated framework for MongoDB cluster setup, leveraging Docker and Kubernetes StatefulSets to streamline configuration, provide persistent storage, and manage replicas. The framework integrates unit testing to validate core MongoDB functionalities and employs fuzz testing with American Fuzzy Lop++ (AFL++) to assess resilience, edge-case handling, and execution path coverage under unpredictable inputs. While experimental results demonstrate automation and the detection of potential vulnerabilities, the scope is limited, necessitating further exploration of complementary testing approaches. Future work will focus on incorporating formal verification methods, advanced performance testing, and fault injection techniques to provide a more comprehensive assessment of MongoDB's reliability in diverse cloud environments. This study highlights the importance of combining multiple testing methodologies to achieve scalable, reproducible, and robust MongoDB deployments.

Related Work

Our project is primarily motivated by two studies: APOLLO, which demonstrates the effectiveness of automated performance regression detection and debugging in database systems, and MongoDB's JavaScript Fuzzer, which highlights the effectiveness of hybrid fuzz testing for uncovering critical bugs in complex database architectures. These works underscore the importance of comprehensive testing methodologies to ensure database systems' reliability, performance, and security in real-world deployments. Inspired by these approaches, our work explores unit and fuzz testing to assess MongoDB's reliability in Chameleon Cloud, addressing gaps in testing scalability and deployment automation.

VM Distribution and Roles

Our project uses Docker and Kubernetes across a three-VM setup to automate the deployment, configuration, and testing of MongoDB clusters. Each VM is designated for specific roles to maximize efficiency, scalability, and reliability.

VM1: Control and Orchestration Node

- Master node in Kubernetes. Deploys all pods.

VM3: Primary MongoDB Replica Set

- Runs Kubernetes pod that hosts the primary MongoDB replica set.

VM4: Secondary MongoDB Replica Set + MongoDB Client

- Runs Kubernetes pod that hosts the secondary MongoDB replica set.
- Runs Kubernetes pod that runs a Python MongoDB client to interact with the MongoDB pods.

Acknowledgements

We would like to thank Professor Aniruddha Gokhale for his invaluable guidance and teachings in the CS 4287/5287 Principles of Cloud Computing course during Fall 2024 at Vanderbilt University.

Test Metrics

1. Functional Metrics

1.1 CRUD Operation Metrics

- Insert Latency (ms): Measure the time taken to insert a single document or batch of documents.
- Query Latency (ms): Evaluate the time taken to retrieve documents based on primary key lookup, range queries, compound key lookups, or text search queries (if indexes are enabled).
- Update Latency (ms): Measure the time to update one or more fields in existing documents.
- Delete Latency (ms): Measure the time to delete documents based on specific criteria.

1.2 Replica Set Metrics

- Replication Lag (ms): Time difference between when a write is applied to the primary node and when it is replicated to secondaries.

2. Performance Metrics

2.1 Throughput

- Operations per Second (ops/s): Measure the number of CRUD operations MongoDB can process per second under a specific workload.

2.2 Resource Utilisation

- CPU Utilisation (%): Measure CPU usage under various workloads.
- Memory Utilisation (%): Track memory usage during read and write-intensive operations.
- Disk I/O (MB/s): Monitor disk reads/writes during heavy data insertions or queries.

3. Reliability Metrics

3.1 Fault Tolerance

- Failover Time (ms): Measure the time taken for the replica set to elect a new primary when the current primary fails.
- Data Consistency: Verify consistency between primary and secondary nodes under normal operations and after failover.

3.2 Durability

- Data Loss on Failure: Measure the extent of data loss (if any) under sudden node or cluster failures.

4. Fuzz Testing Metrics

4.1 Resilience

- Crash Rate (% of operations): Percentage of operations causing crashes during fuzz testing.

4.2 Vulnerability Metrics

- Edge Case Coverage: Number and variety of edge cases detected during fuzz testing.
- Execution Paths Tested: Percentage of code paths executed during fuzz testing.

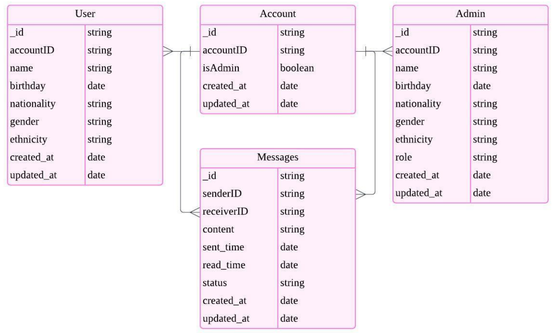
5. Benchmark Metrics

5.1 Load Testing

- Sustained Performance: Measure system performance over an extended period (e.g., 1-hour test).

Database design

Our database schema represents a system for managing users, administrators, accounts, and a messaging platform. Our design supports role-based access control, audit tracking, and a structured messaging system, making it ideal for platforms requiring robust user and admin management.



Future Work

The current work primarily employs unit and fuzz testing to validate MongoDB operations and uncover edge-case vulnerabilities. Future enhancements include integrating formal verification methods to prove the correctness of critical database operations mathematically. For example, model-checking techniques could be employed to verify that MongoDB's replica set configurations conform to consistency and fault-tolerance properties under all possible network conditions. Additionally, fault injection testing could be introduced to simulate real-world failures such as disk crashes, network partitions, or memory exhaustion, allowing the system to evaluate MongoDB's resilience under adverse conditions.

Thus, security and reliability are critical for database systems deployed in cloud environments. Incorporating security testing techniques, such as penetration testing and access control validation, would help identify authentication, authorization, and encryption mechanisms vulnerabilities. Hence we could explore integrating advanced fuzzing tools, such as coverage-guided fuzzing, to achieve higher code path coverage and detect more complex edge cases. Combining these techniques with data consistency checks across primary and secondary replicas would further enhance the system's reliability.

References

1. Jung, Jinho, et al. “APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems.” *Proceedings of the VLDB Endowment*, vol. 13, no. 1, 2020, pp. 57–70, https://doi.org/10.14778/3357377.3357382.

2. Guo, Robert. “MongoDB’s JavaScript Fuzzer: The Fuzzer Is for Those Edge Cases That Your Testing Didn’t Catch.” *ACM Queue*, vol. 15, no. 1, 2017, pp. 38–56, https://doi.org/10.1145/3055301.3059007.

Discussions

Unit testing is well-suited for validating functional metrics such as CRUD operations and replica set configurations. It ensures that MongoDB’s core functionalities behave as expected under controlled conditions. Metrics, like insert latency, query latency, update latency, and delete latency, are effectively covered, as unit tests can be precisely designed to measure these operations under specific scenarios. However, unit testing is inherently limited when addressing metrics that depend on unpredictable or adversarial inputs. For instance:

1. Crash Rate (% of operations): Unit tests are designed to verify expected functionality and do not introduce malformed or randomized inputs. As a result, they fail to expose crashes caused by edge cases or unexpected behaviours.
2. Execution Paths Tested: Unit testing cannot explore diverse and extensive code paths. It focuses on predetermined scenarios, leaving many untested paths untouched, especially those triggered by rare or unexpected conditions.
3. Edge Case Coverage: Unit tests typically cover known scenarios but cannot account for unknown or unexpected input patterns, making them insufficient for identifying obscure bugs.

Fuzz testing complements unit testing by introducing randomised and adversarial inputs into MongoDB operations. This approach excels at identifying vulnerabilities and ensuring robustness under unexpected conditions. Metrics such as crash rate, edge case coverage, and execution paths tested are well-addressed through fuzz testing:

1. Crash Rate (% of operations): Fuzz testing is designed to detect crashes caused by malformed or random inputs, making it highly effective for this metric.
2. Edge Case Coverage: By generating a wide range of inputs, fuzz testing uncovers edge cases that traditional testing approaches often miss.
3. Execution Paths Tested: Coverage-guided fuzzing tools explore diverse code paths, significantly in- creasing the percentage of execution paths tested.

Despite these advantages, fuzz testing is not suitable for certain metrics:

1. Insert, Query, Update, and Delete Latency: Fuzz testing prioritizes randomized inputs over controlled measurement. It cannot reliably capture precise latency metrics under standard operational conditions.
2. Replication Lag: This metric requires precise configurations and synchronization scenarios outside the scope of fuzz testing’s randomized input generation.
3. Throughput (ops/s): Fuzz testing does not focus on sustained performance measurement under realistic workloads, making it unsuitable for benchmarking throughput.