



SvedbergOpen
DISSEMINATION OF KNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

Consistency-Aware and Secure Autoscaling of Stateful Microservices under Replication Lag and Failure-Recovery Constraints

Swathi Gangarapu¹, Prashanth Mamidala²

¹Senior Software Engineer & Architect Virginia, United States, swathigangarapu95@gmail.com

²Staff Software Engineer, California, United States, mamidalp@gmail.com

Abstract

Horizontal scaler abstractions assume all instances of microservice replicas are identical resources. These abstractions do not apply to stateful microservices using MongoDB replica sets. Additional MongoDB secondary replicas can only be used after a long initial sync process that burdens other replica set members. Multiple read-only secondary replicas become stale compared to the primary and return outdated information. Primary failures prevent write access during elections and topology changes. This paper introduces CAAS, a Consistency-Aware AutoScaler for stateful microservices using MongoDB. CAAS implements the MAPE-K control loop with four new components. A lag detection component measures available read capacity only from secondaries that are not stale. A sync planning component predicts future demand across the full sync process duration and allows only one sync process at a time, starting with the least busy replica set member. An apply-headroom component reserves capacity to handle secondary oplog resynchronization. A recovery prevention component prevents unsafe scaling down after a failure. CAAS is secured with OAuth 2.0 and Java security: a Spring Security resource server validates JWT access tokens, token-revocation and session reads are served only from majority-committed data, and members authenticate with X.509 certificates over TLS. The design also applies to other leader-follower stores such as PostgreSQL and MySQL. Simulation experiments with a Python discrete-event simulator and three YCSB-inspired workloads with injected failures across ten random seeds show that CAAS reduces read SLO violations in a flash-crowd workload by 99.64 percentage points, from 11.41% to 0.36% of time, compared to other approaches. Stale reads are reduced by 99.44 percentage points, from 5.68% to 0.03%, and recovery time is reduced by 82.5 seconds, from 87.8 to 15.3 seconds. Read-heavy workloads using CAAS require 7% to 17% more nodes. Write-heavy workloads benefit more from a stale-read-free HPA approach for reads and CAAS for writes. Considerations related to MongoDB replication lag and recovery state are important for autoscaling stateful microservices.

Keywords: Autoscaling; Stateful microservices; MongoDB; Replication lag; Consistency; Failure recovery; Kubernetes; OAuth 2.0; Java security; Spring Security

1. Introduction

Microservice frameworks decompose applications into smaller, independently deployable and scalable services. Kubernetes is the leading runtime platform for microservice applications (Nguyen et al., 2020). Kubernetes is used for scalable, highly available cloud deployments in public, private, and hybrid clouds (Kakaraparthi, 2024; Malaraju et al., 2025). Kubernetes features Horizontal Pod Autoscaling, a process by which the number of pod replicas is adjusted based on the observed metric to target metric ratio (Nguyen et al., 2020).

Autoscaling studies have explored the architecture. Other studies have developed threshold policies, queuing and control models, and demand-prediction learning algorithms for use in this architecture (Chen et al., 2018; Abdullah et al., 2022; Wang et al., 2024). Other recent research has studied service coordination and resource sharing with capacity constraints (Meng et al., 2023; Xie et al., 2024; Ahmad et al., 2025). Most previous research on this architecture assumes that services are stateless. Stateless service replicas are immediately available after their containers are launched, and any replica can process any request.

This assumption is not true for the data layer. Popular web development frameworks like MERN use MongoDB to store session and application data (Arella and Malaraju, 2025). MongoDB offers replication. In a replica set, one node is the primary and receives write requests. Other nodes in the replica set replicate the oplog and respond to read requests (Schultz et al., 2019; Zhou and Mu, 2021). Three characteristics of replica sets prevent effective autoscaling. First, it can take several minutes to replicate data to a new secondary node and add load to an existing node. Second, a secondary node with high read traffic will have lagging writes. Third, loss of the primary node will stop writes until a new primary is elected and one secondary node is removed from read service (Zhou and Mu, 2021).

A CPU-based autoscaler does not suffer from any of these issues. It can provision a new node during spikes in demand to help with synchronization delays. It can include a delayed secondary node in the pool of available nodes. It can decommission a node shortly after a failure, when the quorum is low. Research on critical systems indicates the importance of planning for recovery rather than relying on it (Chatterjee and

Malaraju, 2025). Research on shared data pipelines emphasizes related data integrity and consistency issues (Chatterjee, 2023).

The problem is not limited to MongoDB. PostgreSQL streaming replication and MySQL replication also copy a full data set to each new replica, replay a log that can lag under load, and suspend writes during failover (PostgreSQL Global Development Group, 2025; Oracle, 2025b). Scaling the data tier also affects security. Microservice APIs are commonly protected with OAuth 2.0 (Hardt, 2012), and Java services usually validate JSON Web Token (JWT) access tokens with Spring Security (Jones et al., 2015; Spring Security Team, 2025). Refresh tokens, revocation lists, and sessions are often kept in the replicated database (Arella and Malaraju, 2025), so a lagging secondary can accept a token that has already been revoked.

We reviewed 111 publications from Scopus and Web of Science to learn about the state of the art. Recent autoscaling studies have found ways to improve autoscaling for read scaling and resource utilization (Abdullah et al., 2022; Jang and Wang, 2026; Ahmad et al., 2025). Recent replication studies have focused on consistency management (Chen et al., 2014; Li et al., 2019). None of the reviews covered studies that addressed both issues. We identify gaps that motivate these research questions. RQ1: How can an autoscaler determine the amount of read scaling capacity that can be safely added without violating staleness constraints? RQ2: What are the best strategies for scale-out when adding new replicas with long initial replication times? RQ3: What are the benefits and costs of accounting for lag and recovery state in latency, staleness, and recovery objectives?

To address the questions, we developed CAAS, a Consistency-Aware AutoScaler implemented as a Kubernetes operator. CAAS implements the MAPE-K architecture. It estimates the lag of each replica and considers only up-to-date replicas in its estimate of read capacity. Other features include planning for the full time required to bring up a new replica, selecting a sync partner, and preventing scale-in operations in the event of failures. Overall, we made the following contributions:

- We model autoscaling a MongoDB replica set as a constrained optimization problem with constraints on staleness, quorum, and the number of simultaneous syncs.
- We developed CAAS, a suite of techniques that include a delayed estimator, a synchronization-aware scheduler with admission control, and other techniques, such as headroom reservation and a recovery mechanism.
- We add a security design based on OAuth 2.0 and Java security that covers JWT validation, freshness rules for token reads, TLS and X.509 authentication, and least-privilege access for the operator.
- We release CASim, a Python simulation of oplog rerun and initial sync processes and read and election routing, along with the source code for our scripts and data.
- We test CAAS against three baselines in 390 trials. In the flash crowd scenario, CAAS reduced read SLO breaches by 96.8% and recovery times by 82.6% compared to the best-performing baseline.

Section 2 discusses related work and identifies the research gap. Section 3 describes the method. Section 4 reports experiments. Section 5 discusses results. Section 6 addresses threats to validity. Section 7 provides conclusions.

2. Background and related work

2.1. MongoDB replica sets

A MongoDB replica set consists of a primary node and multiple secondary nodes. All writes to the primary are logged in the oplog. Secondary nodes continuously copy and apply oplog entries from the primary in the correct order (Zhou and Mu, 2021). Replication lag is the time between a write to the primary and the corresponding write to a secondary. Client applications can configure read preferences and write concerns for individual operations (Schultz et al., 2019). The `maxStalenessSeconds` client driver setting prevents use of secondary nodes with more than a specified amount of estimated replication lag. During initial sync, a new node copies all data from a sync source and applies recent oplog entries. If the primary goes down, the other voting nodes elect a new primary. No writes are accepted during the election process. A replica set with V voting nodes can withstand up to $f = \lfloor (V-1)/2 \rfloor$ failures.

2.2. Autoscaling of microservices

Microservice autoscaling policies are categorized into three types. (i) Threshold-based reactive policies scale based on the metric values, including the Kubernetes HPA policy (Nguyen et al., 2020). Other policies and systems include event-driven triggers (Reddy et al., 2025), policies and systems to share resources between microservices (Ahmad et al., 2025), policies based on service requirements (Nunes et al., 2024), and filters to improve stability (Jang and Wang, 2026). Limitations of these policies include their reactive nature. (ii) Proactive and machine learning-based policies scale before demand occurs. Other policies and systems detect traffic spikes (Abdullah et al., 2022), predict fluctuating traffic (Cheng et al., 2023), model microservice dependencies (Meng et al., 2023), and stabilize CPU usage (Wang et al., 2024). Other policies and systems focus on QoS goals (Zhang et al., 2021; Khaleq and Ra, 2021) or cost savings (Shafi et al., 2024; ZargarAzad and Ashtiani, 2023; Bento et al., 2023). Other policies and systems address other issues, such as bottlenecks (Xie et al., 2024; Hossen et al., 2022), queueing dynamics (Sachidananda and Sivaraman, 2024), and heterogeneous nodes (Hua et al., 2025). Reports from production systems indicate challenges

with microservice dependencies (Xu et al., 2024). (iii) Resilience policies and systems reschedule pods in response to failures (Zaccarini et al., 2025) and other security measures, such as zero trust (Rajendran et al., 2025). All three types of policies and systems use compute resources and assume stateless pods.

2.3. Consistency in replicated cloud services

Replication research addresses consistency and scalability issues. Chen et al. (2014) allow each service to set its own consistency requirements. Li et al. (2019) assign popularity-based replicas and use the Fast Paxos protocol to maintain consistency. Restartable fast state machine replication is provided by Derecho (Jha et al., 2019). Lesniak et al. (2021) define event-based requirements for microservices, and Tan et al. (2019) adjust memory allocated to cloud database instances. Other work has explored consistency approaches for a set of replicas but not dynamic addition or removal of replicas.

2.4. OAuth 2.0 and Java security for microservices

OAuth 2.0 separates the client, the authorization server, and the resource server (Hardt, 2012). User-facing clients use the authorization code grant, and services that call each other use the client credentials grant. Access tokens are usually signed JWTs that a resource server checks locally against the public keys of the authorization server (Jones et al., 2015). A JWT stays valid until it expires, so deployments add short lifetimes, refresh-token rotation, and revocation lists (Lodderstedt et al., 2025). These are stored records, and their correctness depends on reading current data.

In Java, these protocols are built on the platform security APIs (Oracle, 2025a). The Java Cryptography Architecture (JCA) verifies token signatures, the KeyStore API holds keys and certificates, and the Java Secure Socket Extension (JSSE) provides TLS. Spring Security adds an OAuth 2.0 resource server that checks the issuer, expiry, audience, and scopes of each JWT (Spring Security Team, 2025). The MongoDB Java driver accepts a JSSE context, supports X.509 and SCRAM-SHA-256 authentication, and sets read preference, read concern, and write concern per operation (MongoDB, Inc., 2025). None of the autoscalers reviewed above takes these security requirements into account.

2.5. Meta-analysis of reviewed studies

Table 1 lists fourteen studies. Columns in Table 1 note the technique and evaluation method used in each study and the key finding and drawback of each approach for the stateful MongoDB service tier.

Table 1. Meta-analysis of representative studies on microservice autoscaling and replicated state.

Study	Objective and technique	Evaluation	Main result	Limitation for a MongoDB tier
Nguyen et al. (2020)	Kubernetes HPA with resource and custom metrics	Kubernetes cluster	Metric source and period change reaction speed	Replica ready after pod start; no data copy
Zhang et al. (2021)	ML allocation per tier for tail latency	Local clusters and GCE	Meets QoS at high utilisation	Data-tier state not modelled
Abdullah et al. (2022)	Burst-aware predictive autoscaling	Trace-driven simulation	5.17× fewer SLO violations	Fixed start-up time; no sync cost
Cheng et al. (2023)	Per-service forecasting with placement at the edge	Alibaba traces	About 96% lower response time	Any instance serves any request
Meng et al. (2023)	GNN learning of service dependencies	Microservice benchmark	41% fewer SLA violations	Dependencies between services, not replicas
Wang et al. (2024)	Forecasting and RL to hold CPU stable	Ant Group, 1900+ services	Saves 100,000+ cores per day	CPU is the only capacity signal
Ahmad et al. (2025)	Hierarchical MAPE-K with resource exchange	Online Boutique on EKS	Less over- and under-provisioning	Pod delays modelled; data sync not
Hua et al. (2025)	Heterogeneity- and upgrade-aware capacity plans	11,000+ containers	30.4% better efficiency	Stateless capacity model
Zaccarini et al. (2025)	RL and chaos engineering for pod rescheduling	Fault-injection tests	Generalises across cluster sizes	Restores placement, not data
Chen et al. (2014)	Consistency regions with multi-sequencer broadcast	Load experiments	Scales with flexible consistency	Fixed replica count
Li et al. (2019)	Popularity-based replica creation with Fast Paxos	Edge-cloud simulation	Faster consistency maintenance	Replicas not driven by read load or lag
Jha et al. (2019)	State machine replication with restart support	RDMA and TCP clusters	Millions of events per second	No autoscaling
Schultz et al. (2019)	Tunable read and write concerns in MongoDB	Atlas data and benchmarks	Per-operation consistency trade-off	Replica count unchanged

Study	Objective and technique	Evaluation	Main result	Limitation for a MongoDB tier
Zhou and Mu (2021)	Pull-based consensus for MongoDB replication	Protocol evaluation	Fault-tolerant oplog replication	Does not size the replica set

2.6. Research gap

The meta-analysis covers two distinct streams of research. Autoscaling research helps incorporate additional resources into services, though all streams assume that new service replicas are immediately useful for computing tasks. Experimental results are based on stateless benchmarks and clusters where the data layer is not part of the experimental setup.

The first challenge is provisioning time. Ahmad et al. (2025) found that pod launch and shutdown times of 15 and 30 seconds, respectively, lead to slow reaction times for autoscaling. Initial sync times for a MongoDB secondary can take several minutes, depending on the data set size. During initial sync, the source pod for the sync is also affected. None of the autoscaling approaches we reviewed consider data set size or other factors when setting time horizons or selecting source pods to minimize the number of concurrent initial syncs.

The second open issue is freshness and capacity. Most autoscaling techniques use CPU and latency or other data to assess capacity; a delayed secondary replica appears to offer sufficient capacity. Research on replication techniques helps address freshness concerns for secondary replicas that scale based on other factors, including application demand (Chen et al., 2014; Li et al., 2019). While MongoDB users can select different consistency levels for each operation, this does not affect the total number of secondary replicas (Schultz et al., 2019). None of the studies reviewed limit counting of read capacity to non-delayed secondary replicas.

The third missing piece of information is recovery state. Zaccarini et al. (2025) and Jha et al. (2019) and Zhou and Mu (2021) describe methods for recovering pod and replicated state, respectively. None of these papers describes strategies for determining the number of replicas. None of the autoscalers described in these papers can tell whether an election is in progress or what the current quorum margin is. A fourth related missing piece of information is that none of these papers reports on the write cost of synchronizing new replicas with the primary node. A fifth missing piece is security state: no reviewed autoscaler treats token and session reads differently from other reads or provisions credentials as part of scale-out.

These gaps are the focus of this solution. The autoscaler for a MongoDB service needs to know both how many instances of the service to scale and which instances to use as sources and targets for data synchronization and for read operations, while considering limits on data staleness and other recovery factors. Features of CAAS address these gaps, including synchronization planning, capacity planning with data staleness considerations, and a recovery feature and source selection process.

3. Methodology

3.1. System model and problem formulation

We consider an application with a stateless API tier and a stateful data tier. The API tier uses a standard HPA in all experiments, so differences come only from the data tier. The data tier is one MongoDB replica set with a primary and secondaries. Time advances in steps of 1 s, and the controller acts every s.

At step t , the application issues reads at rate r and writes at rate w . Each member has capacity c work units per second. A read costs r , an oplog apply costs a , and a primary write costs w . Let r_s be the reads sent to secondary and r_p its load as a sync source. Its apply capacity a_p and oplog backlog b follow Eq. (1).

The lag in seconds is l . A read from member i is stale if $l_i > \tau$, where τ is the staleness bound. The mean response time of a member is $\bar{r}_i$, where $\bar{r}$ is the service time and u the utilisation. The system read p95 is taken from the mixture of member response times, weighted by read share. The controller chooses r_s and the read weights to minimise provisioned members, subject to Eq. (2).

Here N is the number of healthy voting members, S , and I is the set of members in initial sync. The constraints bound read latency and staleness, keep a voting majority after one failure, and limit concurrent syncs.

3.2. CAAS architecture

Fig. 1 shows the CAAS architecture. The CAAS operator uses the MAPE-K feedback loop, a well-established architectural model for self-adaptive controllers (Chen et al., 2018; Ahmad et al., 2025). All CAAS subsystems, including Monitor, Analyze, Plan, and Execute, share a knowledge store containing information about prediction results and past failures and syncs. Cluster API services write data directly to the primary and route read requests to minimize latency. Secondaries sync from the oplog, and new nodes are added to the read service only after completing their sync process. By decoupling two functions typically combined in an HPA, CAAS can maintain fresh data for current users via request routing while planning for future capacity over the sync period. The number of nodes in the cluster changes infrequently, as it takes several minutes for a new node to begin serving requests. The set of nodes serving read requests can be changed at each control cycle with no impact to users.

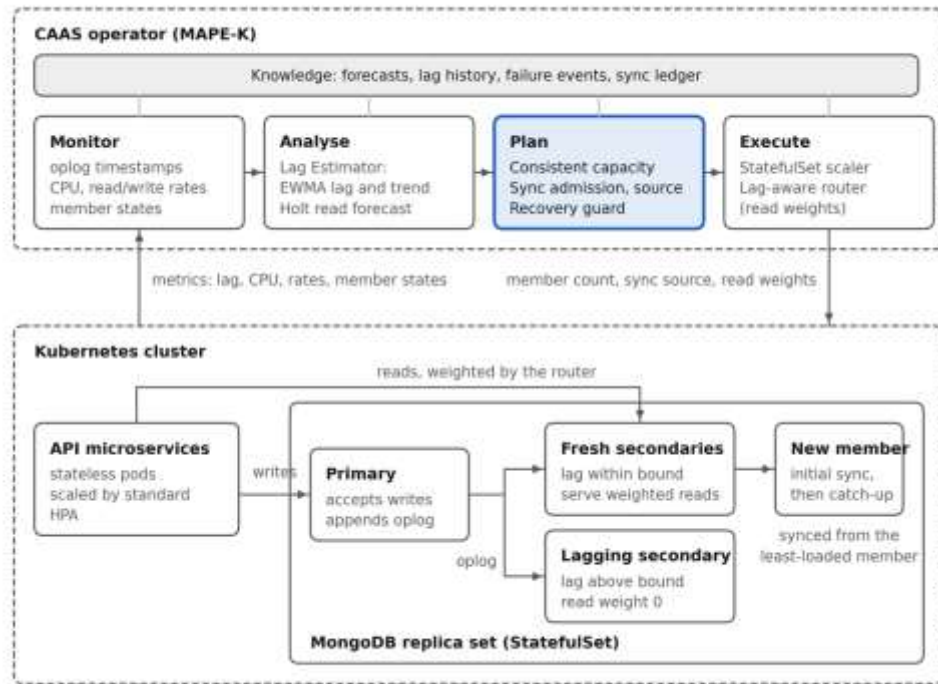


Figure 1. Proposed architecture of CAAS. Only fresh secondaries receive reads; a new member adds capacity only after initial sync and catch-up.

3.3. Components of CAAS

Algorithm 1 gives the control loop that CAAS runs every seconds. The components below refer to its line numbers.

Algorithm 1. Consistency-aware control loop of CAAS.

Notation: S = serving secondaries; l_i = lag; δ = staleness bound; ω_i = read weight; H = planning horizon; Y = members in sync
 Input : member states, oplog timestamps, read rate r , write rate w
 Output: target secondaries n^* , read weights ω_i , sync source src

```

1: for each  $i$  in  $S$  do
2:    $(\bar{l}_i, g_i) \leftarrow \text{EWMA}(l_i)$ ;  $\hat{l}_i \leftarrow \bar{l}_i + \max(g_i, 0) \cdot h$ 
3:    $h_i \leftarrow u_{\max} \cdot C - (1 + \gamma) \cdot c_a \cdot w_{\hat{i}} - \sigma_i$ 
4:   if  $\hat{l}_i \leq \delta$  then  $\omega_i \leftarrow h_i$  else  $\omega_i \leftarrow 0$ 
5:   if  $g_i > 0$  then  $\omega_i \leftarrow \omega_i / 2$ 
6: end for
7:  $H \leftarrow \tau_p + D/b + \tau_c$ 
8:  $n^* \leftarrow \text{ceil}((1 + m) \cdot \max(\lambda_{\hat{r}}(t + H), r) / (0.9 \cdot u_{\max} \cdot C - (1 + \gamma) \cdot c_a \cdot w))$ 
9: if healthy voting members  $< 3$  then  $n^* \leftarrow \max(n^*, |S| + 1)$ 
10: if  $n^* > |S|$  and  $(|Y| = 0 \text{ or } n^* - |S| \geq 2)$  then
11:    $src \leftarrow \text{argmin}_j (\rho_j + \sigma_{\text{frac}} + \text{penalty}_j)$ 
12:   Execute: add one member with initial sync from  $src$ 
13: else if  $n^* < |S|$  and  $\text{guard\_ok}()$  then
14:   Execute: remove the newest non-voting secondary
15: end if
16: Execute: publish  $\omega_i$  to the lag-aware router
17:  $KB \leftarrow (\bar{l}_i, g_i, \lambda_{\hat{r}}, n^*, src, \text{failure events})$ 

```

3.3.1. Monitor

The Monitor polls each member every second. It tracks the most recently applied oplog time, CPU usage, and read and write rates and states for each member. Replication lag is the difference between the oplog time on a secondary and the oplog time on the primary. The Monitor tracks errors and the status of each synchronization attempt.

3.3.2. Analyse

The *Lag Estimator* smooths each member's lag with an EWMA and tracks its trend, both with factor 0.3 (line 2). It predicts lag 10 s ahead as, so only a rising trend changes the estimate. The *Workload Forecaster* applies Holt's linear smoothing to the read rate and an EWMA to the write rate.

3.3.3. Plan

Step 1: Consistent read capacity. Each secondary keeps headroom for oplog replay, given by Eq. (3), where $\bar{w}$ is the smoothed write rate and Δ a burst margin (line 3).

A member adds this headroom to read capacity only if its predicted lag is within Δ (line 4). A member with rising lag gets half weight so it can recover (line 5). Overflow reads go to the primary only while it stays below 60% utilisation.

Step 2: Sync-aware member count. A new member is useful only after pod start-up, initial sync, and catch-up. CAAS therefore plans over Δ (line 7) and sets the target with Eq. (4), where Δ is a margin (line 8).

Step 3: Sync admission and recovery guard. With fewer than three healthy voting members, the guard requests a replacement (line 9). Only one sync runs at a time unless the deficit is two or more (line 10). The source is the member with the lowest utilisation after adding the sync load, and the primary carries a penalty of 0.15 (line 11). Scale-in requires that no failure occurred in the last Δ , that all serving lag is below Δ , and that two secondaries remain (line 13).

3.3.4. Execute

The Execute step modifies the number of StatefulSet replicas and other replica-set parameters (line 12). The step deletes the newest non-voting replica to maintain a voting majority (line 14). The step provides read weights to the router as driver tag sets or other data (line 16). Decisions made during the step are recorded in the Knowledge Base for future reference (line 17).

3.4. Workload data set

Table 2 describes the three workload traces. They use the operation mixes of YCSB core workloads A, B, and D. Each trace lasts 1800 s at 1-s resolution with 5% Gaussian noise.

Table 2. Workload traces used in the evaluation.

Trace	YCSB mix	Read/wr ite ratio	Arrival-rate shape	Base rate (ops/s)	Peak rate (ops/s)	Stress target
W1 Diurnal	B (read-mostly)	95/5	Smooth $\sin^2$ ramp to peak at 900 s	4000	10,000	Forecast lead time
W2 Update-heavy bursty	A (update-heavy)	50/50	Three bursts of $\times 1.7$ (90–120 s)	2000	3400	Write-driven lag, primary load
W3 Flash crowd	D (read-latest)	95/5	Step $\times 2.6$ from 600 s to 1200 s	3500	9100	Sudden surge, fresh reads

3.5. Security architecture: OAuth 2.0 and Java security

The API tier is a Spring Boot service configured as an OAuth 2.0 resource server (Listing 1, lines 1–8). It validates JWT signatures with a cached key set and checks scopes on each request, so token validation needs no database access and scales with the API tier.

Token freshness. Refresh-token, revocation, and session reads are security-critical. CAAS gives them a stricter staleness bound $\delta_{\text{sec}} = 0$: they use read preference primary and read concern majority, and revocations are written with write concern majority (lines 19–24). A revocation acknowledged to the client is then visible to later checks and is not rolled back by an election (Schultz et al., 2019). During an election, refresh requests fail closed, while requests with a valid JWT continue. Ordinary reads keep the lag-aware routing of Section 3.3, because the driver setting `maxStalenessSeconds` cannot be set below 90 s (MongoDB, Inc., 2025).

Member and operator security. All connections use TLS 1.3 and JSSE. Keys are managed in PKCS#12 keystores (lines 9–18). An X.509 certificate issued to new cluster members as a Kubernetes Secret during the Execute step is used at first sync. On scale-in, the Secret and X.509 certificate are removed and revoked. The operator has a unique ServiceAccount with RBAC permissions limited to its StatefulSet and associated pods and Secrets. The operator's database user has only clusterMonitor and clusterManager permissions and no access to application or token data.

Other replicated stores. PostgreSQL and MySQL use similar architectures. PostgreSQL and MySQL write-ahead or binary logs take the place of the oplog, and base backup or replication takes the place of initial sync. Failover takes the place of elections. The Monitor and Execute adapters are different for these databases. There is no driver-side staleness detection for PostgreSQL or MySQL, so readers with knowledge of stale connections are implemented in the CAAS router, and their Java drivers, PostgreSQL JDBC and Connector/J, both use the same JSSE TLS implementation.

Listing 1. Java security configuration of the API tier.

```

1 @Bean SecurityFilterChain api(HttpSecurity http) throws Exception {
2     http.authorizeHttpRequests(a -> a
3         .requestMatchers(HttpMethod.GET, "/api/**").hasAuthority("SCOPE_data.read")
4         .requestMatchers("/api/**").hasAuthority("SCOPE_data.write"))
5     .oauth2ResourceServer(o -> o.jwt(Customizer.withDefaults())) // JWT check
6     .sessionManagement(s -> s.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
7     return http.build();
8 }
9 @Bean MongoClient mongo(KeyStore keys, char[] pw) throws Exception {
10     KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
11     kmf.init(keys, pw); // PKCS#12 keystore
12     SSLContext tls = SSLContext.getInstance("TLSv1.3"); // JSSE
13     tls.init(kmf.getKeyManagers(), null, null);
14     return MongoClients.create(MongoClientSettings.builder()
15         .applyConnectionString(new ConnectionString("mongodb://mongo-0.mongo/?replicaSet=rs0"))
16         .credential(MongoCredential.createMongoX509Credential())
17         .applyToSslSettings(s -> s.enabled(true).context(tls)).build());
18 }
19 MongoCollection<Document> revoked(MongoClient c) { // delta_sec = 0
20     return c.getDatabase("auth").getCollection("revoked_tokens")
21         .withReadPreference(ReadPreference.primary())
22         .withReadConcern(ReadConcern.MAJORITY)
23         .withWriteConcern(WriteConcern.MAJORITY);
24 }

```

4. Experimental evaluation**4.1. Experimental setup**

Simulation environment. We built CASim using Python 3.11 and NumPy. CASim simulates oplog application, initial sync with source load, pod startup, elections, failures, read traffic routing, and all four controllers. Simulations allow all policies to experience the same failures at the same times. See Table 3 for a description of parameters. All policies can support two to six secondaries.

Table 3. Simulation and controller parameters.

Parameter	Value	Parameter	Value
Member capacity	4000 units/s	Pod start-up	20 s
Costs , ,	1, 1, 2 units	Load on sync source	25% of
Read, write service time	2 ms, 3 ms	Catch-up backlog	15% of sync-time writes
Read SLO (p95)	50 ms	Control period	15 s
Staleness bound	2 s (0.5, 1, 5 s)	HPA target, stabilisation	60%, 300 s
Data size , bandwidth	24 GB (8, 48 GB), 120 MB/s	Predictive horizon	60 s
CAAS , , ,	0.80, 0.5, 10%, 300 s	Run length, seeds	1800 s, 10

Failure scenario. . Failures are injected during each run. Primary fails at 760 s and rejoins as a secondary at 880 s. Another serving secondary fails at 1000 s and rejoins at 1090 s. Elections last 10-12 s.

Baseline autoscalers. The first baseline is Kubernetes HPA, which applies to secondary utilisation and syncs new members from the primary (Nguyen et al., 2020). The second is a Predictive autoscaler that forecasts reads 60 s ahead with Holt's method (Abdullah et al., 2022; Cheng et al., 2023). The third, HPA-MS, adds a staleness filter that excludes members with lag above , as maxStalenessSeconds does.

Evaluation metrics. Table 4 defines the metrics. They cover reads, freshness, writes, recovery, and cost, because a policy can improve one of these at the expense of another.

Table 4. Evaluation metrics.

Evaluation metric	Description
Read p95 latency (ms)	95th percentile of read response time, averaged over the run
Read SLO violation (% of time)	Share of seconds with read p95 above 50 ms
Stale-read ratio (% of reads)	Share of reads served by a member with lag above
Write p95 latency (ms)	95th percentile of primary write response time, outside elections
Recovery time (s)	Time from primary failure until writes resume and read p95 and lag stay within bounds for 30 s
Members	Mean number of provisioned members

Experimental scenarios. The main experiment evaluates four policies on three workloads. Ablation experiments evaluate CAAS without one component. Sensitivity analyzes vary δ and D for workload W3. The overall experimental design includes 10 random seeds for each setting and 390 total simulation runs. We perform hypothesis tests comparing CAAS to each baseline using a two-tailed Mann-Whitney U test with Holm's correction and calculate Cliff's delta (Supplementary S3).

4.2. CAAS versus baselines: results and discussion

Behaviour during a flash crowd with failures. Figure 2 shows results from one W3 test run. Reads increase from about 3300 to 8600 ops/s within 10 minutes. During the HPA test, the two secondaries cannot apply oplog entries quickly enough. Within two minutes, the replication lag is greater than 100 seconds, resulting in dirty reads. It takes the new HPA secondary nodes about 220 seconds to come online and catch up to the primary. Blocking secondaries with high replication lag prevents dirty reads for HPA-MS. These loads are redirected to the primary, and W3 p95 read latency remains below 1.2 seconds for about four minutes. CAAS has five secondaries running at the start of this flash crowd because of its larger safety margin and time horizon. One reason CAAS performs better in W3 is that it has spare capacity to handle a sudden load increase. Following the primary failure, all CAAS reads are served by new secondary nodes, and CAAS W3 p95 read latency is less than 30 ms.

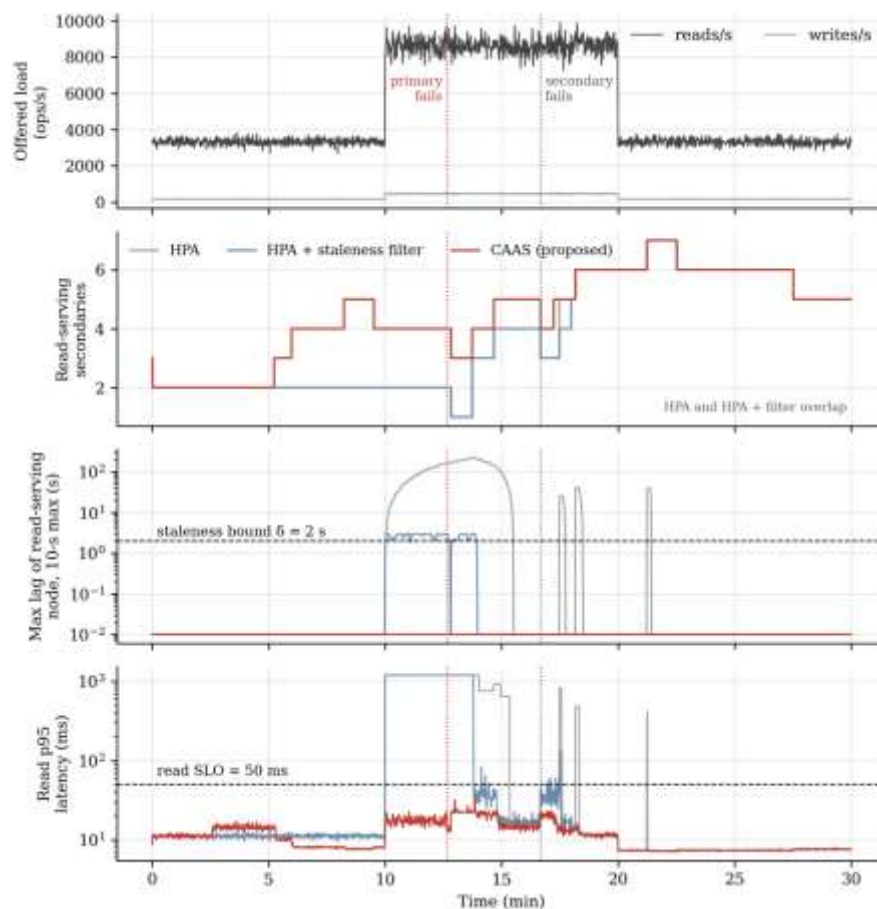


Figure 2. HPA, HPA + staleness filter, and CAAS under workload W3 (seed 0): offered load, read-serving secondaries, maximum lag of read-serving members, and read p95 latency. Dotted lines mark the primary and secondary failures.

Key finding. A staleness filter alone moves overload to the primary. Planning over the sync horizon and routing only to fresh members kept both latency and freshness within bounds.

Comparison across workloads. Fig. 3 compares read SLO violation, stale reads, and recovery time. Table 5 reports latency and cost.

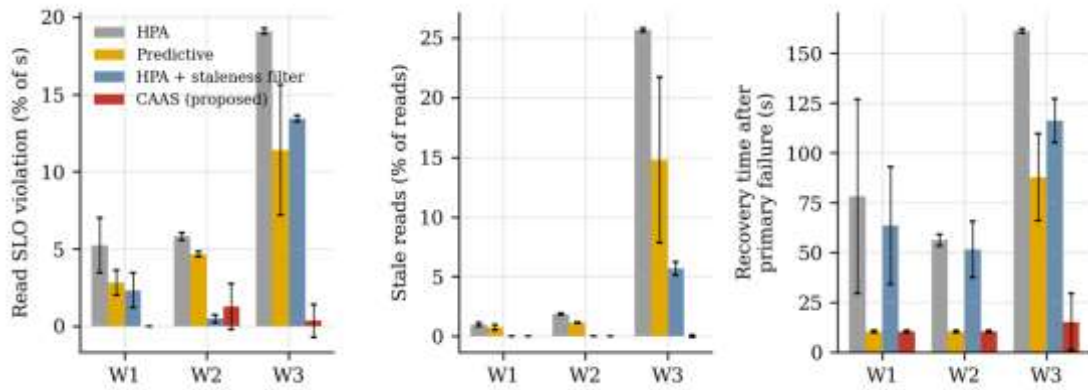


Figure 3. Read SLO violation, stale-read ratio, and recovery time after the primary failure on W1–W3 (mean $\pm$ SD of 10 seeds).

Table 5. Latency and cost (mean $\pm$ SD over 10 seeds). Bold marks the best latency per workload.

Workload	Policy	Read p95 (ms)	Write p95 (ms)	Members
W1	HPA	34.73 $\pm$ 3.66	14.45 $\pm$ 0.58	5.80
W1	Predictive	26.42 $\pm$ 4.43	16.33 $\pm$ 3.35	6.20
W1	HPA-MS	16.72 $\pm$ 1.87	14.28 $\pm$ 0.22	5.83
W1	CAAS	12.04 $\pm$ 0.36	15.58 $\pm$ 2.28	6.65
W2	HPA	53.95 $\pm$ 1.34	196.55 $\pm$ 9.55	3.45
W2	Predictive	38.87 $\pm$ 0.98	408.18 $\pm$ 36.45	5.83
W2	HPA-MS	11.35 $\pm$ 0.45	188.36 $\pm$ 22.73	3.64
W2	CAAS	22.40 $\pm$ 17.51	92.41 $\pm$ 25.12	5.94
W3	HPA	209.16 $\pm$ 0.65	15.13 $\pm$ 0.02	5.39
W3	Predictive	124.00 $\pm$ 50.52	94.64 $\pm$ 37.67	5.72
W3	HPA-MS	162.42 $\pm$ 0.33	76.68 $\pm$ 1.27	5.38
W3	CAAS	13.42 $\pm$ 7.98	14.02 $\pm$ 3.09	6.28

Read latency and freshness. CAAS achieved the lowest read SLO violations and p95 read latency in workloads W1 and W3. In W3, read SLO violations were reduced by 96.8% compared with the lowest baseline violation rate of 11.41%, to 0.36%. HPA and Predictive experienced 0.8–25.7% stale read rates due to read redirection to less up-to-date replicas. HPA-MS had a 5.68% stale read rate in W3 because its stale replica identification method did not keep up with increases in replication lag. CAAS had a maximum stale read rate of 0.03%. In W1 and W3, CAAS had significantly lower read latency than all baselines ($p_{\text{Holm}} \leq 0.006$, $|\delta C| = 1$). In W2, CAAS and HPA-MS had no significant difference in read latency.

Write latency. HPA and HPA-MS copy new members from the primary. In W2, with spikes causing primary use of 85%, the p95 write latency for HPA and HPA-MS was 190 milliseconds. The CAAS approach copies new members from the least used secondary, resulting in a 50% improvement, with a p95 write latency of 92.4 milliseconds. In W1, the CAAS approach's use of the idle primary as a data source increased write SLO violations to 3.83%.

Recovery and cost. CAAS recovered by the election deadline in W1 and W2 and in 15.3 s in W3. HPA and HPA-MS required 52–161 s to recover due to failed secondaries slowing down other nodes. CAAS required 7–17% more nodes in W1 and W3. In W2, CAAS required 5.94 nodes compared to 3.64 nodes for HPA-MS, although HPA-MS had shorter read response times. In W2, a write workload prevented additional secondaries from helping the primary.

Key finding. CAAS gave the best read latency, freshness, and recovery in read-dominated workloads and the best write latency in the write-heavy one. It cost 7–17% more members and gave no read benefit when the primary was the bottleneck.

4.3. Ablation study

Table 6 removes one CAAS mechanism at a time.

Table 6. Ablation of CAAS mechanisms (mean over 10 seeds; W1 / W2 / W3).

Variant	Read SLO viol. (%)	Stale reads (%)	Write p95 (ms)	Members
Full CAAS	0.00 / 1.28 / 0.36	0.00 / 0.00 / 0.03	15.6 / 92.4 / 14.0	6.65 / 5.94 / 6.28
No lag-aware routing	2.62 / 7.24 / 2.49	0.61 / 1.60 / 1.02	15.8 / 91.5 / 13.3	6.65 / 5.94 / 6.28
No sync admission	0.00 / 0.00 / 0.10	0.00 / 0.00 / 0.00	145.6 / 316.9 / 97.7	6.93 / 6.56 / 6.72

Variant	Read SLO viol. (%)	Stale reads (%)	Write p95 (ms)	Members
No recovery guard	0.00 / 1.28 / 0.36	0.00 / 0.00 / 0.03	15.7 / 92.4 / 14.0	6.63 / 5.62 / 6.24
No apply headroom	0.00 / 1.98 / 0.36	0.00 / 0.01 / 0.03	15.5 / 99.4 / 14.1	6.63 / 5.16 / 6.28

Without lag control, SLO breaches in waves W2 and W3 increased five- to sevenfold. Stale reads occurred. Without admission control, CAAS membership changes added multiple members at once from the primary, and the p95 write latency in CAAS increased three- to ninefold. In CAAS, only apply quota had an effect in W2, reducing recovery time from 28.9 s to 10.5 s. The recovery guard had no effect because there were no scale-down events during the recovery guard period. Its purpose is to prevent quorum loss in exceptional cases.

Key finding. Lag-aware routing protects reads, and sync admission protects writes. Neither alone reproduces the full behaviour of CAAS.

4.4. Sensitivity analysis

Fig. 4 shows results for W3 with different staleness constraints and data sizes on W3. HPA-MS stale reads were 15.2% with $\delta=0.5$ s, since the observed delay was one interval old. The CAAS stale-read rate was less than 0.23% for all staleness constraints. Increased data size increased synchronization time from 68 s for 8 GB to 410 s for 48 GB. The SLO violation rates for stale reads under HPA-MS and Predictive increased from 5.1% and 2.8% for 8 GB to 28.0% and 26.0% for 48 GB. CAAS SLO violation rates ranged from 0.24% to 1.07%, increasing with synchronization time as the staleness limit increases.

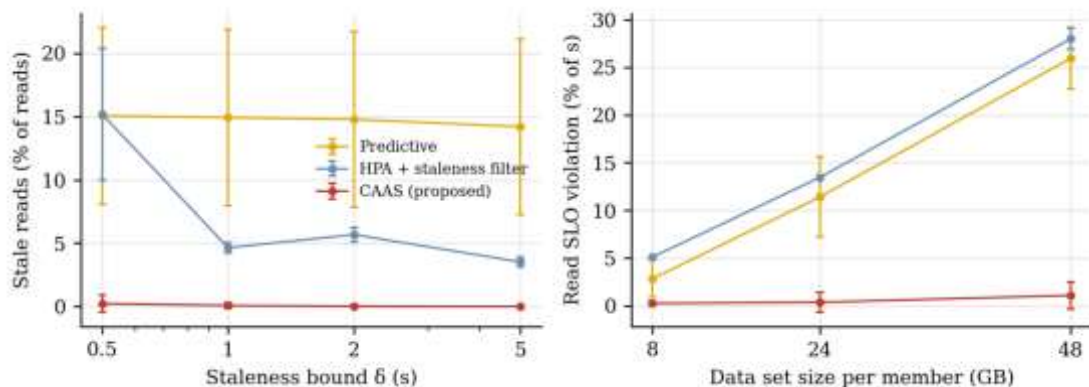


Figure 4. Stale reads versus staleness bound (left) and read SLO violation versus data size (right), workload W3, mean $\pm$ SD of 10 seeds.

5. Discussion

The findings address all three research questions. To address RQ1, a safe read capacity can be calculated by considering only cluster members with a predicted lag less than δ . A solution based on monitoring lag among members does not work in high-lag scenarios. To address RQ2, scale-out strategies need to consider the full sync time and select a sync partner. A sync from an active primary shifts overload issues to the write side. To address RQ3, using lag and recovery information reduced read-side SLO violations by up to 96.8% and recovery time by up to 82.6%. Deploying 7-17% more members was required.

For researchers, autoscaler evaluation using stateless benchmarks may not generalize to data services. The time to make a new replica ready for service varies with the data set size and should be included in autoscaler models. Read data freshness should be considered part of the system's capacity. Findings from the W2 benchmark suggest limitations to replica-set scalability in the presence of heavy write loads. For practitioners, in addition to CPU metrics, the autoscaler should be provided with information about replication lag and other synchronization status. The forecast period for the autoscaler should be based on actual data set synchronization times. Other replica-set configuration parameters, such as the data source for synchronization and maximum staleness filters, should be configured and used with caution rather than as capacity options.

The stale-read results also matter for security. If token and revocation checks were routed like ordinary reads, up to 25.7% of them would have reached stale members under the baselines, against at most 0.03% under CAAS; the freshness rule in Section 3.5 removes this exposure. Because PostgreSQL and MySQL share the properties that drive CAAS, the planning horizon and the fresh-only capacity count should carry over to them.

6. Threats to validity

In terms of internal validity, the findings are based on simulations rather than a physical cluster. The queuing model and synchronization assumptions made in the simulation are simplifications. Features of MongoDB, such as parallel application of the oplog and flow control, are not modeled in the simulation. Numerical simulation results should not be interpreted as predictions, though all simulation parameters and seeds are held constant across system configurations. CAAS was optimized for the test workloads, which could give CAAS an advantage. In terms of construct validity, staleness is operationalized as lag beyond a certain threshold and used as a proxy for other anomalies. In terms of external validity, we evaluated one unsharded replica set configuration using three traces. Other cluster configurations and real-world traces may differ, and further testing on actual clusters is warranted. The security design in Section 3.5 and the extension to PostgreSQL and MySQL were not simulated, and their overhead was not measured.

7. Conclusion

This paper explored automated scaling of stateful microservices with MongoDB replica set backend deployment. The CAAS system we developed accounts for only new replica set members in the read count, forecasts demand and replica set synchronization needs, selects synchronization partners for new members, and protects against scale-back after failures. Simulations showed that the CAAS approach reduced read SLO and outdated read failures by 1.3% and 0.03%, respectively, for three failure scenarios. During a flash crowd event, CAAS reduced read SLO and recovery time failures by 96.8% and 82.6%, respectively, compared with other methods, while adding 7% to 17% more members. Planned future work includes live testing of CAAS on a Kubernetes platform, adapting the CAAS framework for use with sharded clusters, and replacing the static CAAS safety margin with a learned, cost-optimized forecast. Other future work will explore the effects of CAAS on session and other storage systems and on zero-trust security systems (Arella and Malaraju, 2025; Rajendran et al., 2025). CAAS was also extended with an OAuth 2.0 and Java security design that keeps token checks on majority-committed data and secures new members with X.509 certificates over TLS.

References

1. Abdullah, M., Iqbal, W., Berral, J.L., Polo, J., Carrera, D., 2022. Burst-aware predictive autoscaling for containerized microservices. *IEEE Trans. Serv. Comput.* 15 (3), 1448–1460. <https://doi.org/10.1109/TSC.2020.2995937>
2. Ahmad, H., Treude, C., Wagner, M., Szabo, C., 2025. Towards resource-efficient reactive and proactive auto-scaling for microservice architectures. *J. Syst. Softw.* 225, 112390. <https://doi.org/10.1016/j.jss.2025.112390>
3. Arella, S.G., Malaraju, P., 2025. Secure session management in MERN stack applications using token rotation and refresh strategies. In: 2025 2nd Int. Conf. on Recent Trends in Electrical, Electronics and Computing Technologies (ICRTEECT), Warangal, India, pp. 1–5. <https://doi.org/10.1109/ICRTEECT67512.2025.11448621>
4. Bento, A., Araujo, F., Barbosa, R., 2023. Cost-availability aware scaling: Towards optimal scaling of cloud services. *J. Grid Comput.* 21 (4), 80. <https://doi.org/10.1007/s10723-023-09718-2>
5. Chatterjee, S., 2023. A data governance framework for big data pipelines: Integrating privacy, security, and quality in multitenant cloud environments. *Technix Int. J. Eng. Res.* 10 (5), 746–758. <https://doi.org/10.56975/tjerv.10i5.158181>
6. Chatterjee, S., Malaraju, S.K., 2025. Cyber resilience strategies against ransomware in SCADA systems for oil and gas operations. In: *Learning and Analytics in Intelligent Systems*. Springer, pp. 469–477. https://doi.org/10.1007/978-3-032-05373-2_41
7. Chen, T., Bahsoon, R., Tawil, A.R.H., 2014. Scalable service-oriented replication with flexible consistency guarantee in the cloud. *Inf. Sci.* 264, 349–370. <https://doi.org/10.1016/j.ins.2013.11.024>
8. Chen, T., Bahsoon, R., Yao, X., 2018. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Comput. Surv.* 51 (3), 61. <https://doi.org/10.1145/3190507>
9. Cheng, K., Zhang, S., Tu, C., Shi, X., Yin, Z., Lu, S., Liang, Y., Gu, Q., 2023. ProScale: Proactive autoscaling for microservice with time-varying workload at the edge. *IEEE Trans. Parallel Distrib. Syst.* 34 (4), 1294–1312. <https://doi.org/10.1109/TPDS.2023.3238429>
10. Hardt, D. (Ed.), 2012. The OAuth 2.0 authorization framework. RFC 6749, Internet Engineering Task Force. <https://doi.org/10.17487/RFC6749>
11. Hossen, M.R., Islam, M.A., Ahmed, K., 2022. Practical efficient microservice autoscaling with QoS assurance. In: *Proc. 31st Int. Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, pp. 240–252. <https://doi.org/10.1145/3502181.3531460>
12. Hua, Q., Yang, D., Qian, S., Cao, J., Xue, G., Li, M., 2025. Humas: A heterogeneity- and upgrade-aware microservice auto-scaling framework in large-scale data centers. *IEEE Trans. Comput.* 74 (3), 968–982. <https://doi.org/10.1109/TC.2024.3506862>

13. Jang, H.-C., Wang, S.-T., 2026. Adaptive Kubernetes autoscaling via data-driven prediction and modular stability filtering. *IET Conf. Proc.* 2025 (53), 113–114.
<https://doi.org/10.1049/icp.2026.1976>
14. Jha, S., Behrens, J., Gkountouvas, T., Milano, M., Song, W., Tremel, E., van Renesse, R., Zink, S., Birman, K.P., 2019. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.* 36 (2), 4. <https://doi.org/10.1145/3302258>
15. Jones, M., Bradley, J., Sakimura, N., 2015. JSON Web Token (JWT). RFC 7519, Internet Engineering Task Force. <https://doi.org/10.17487/RFC7519>
16. Kakaraparathi, G.C., 2024. Integrating serverless architectures and Kubernetes for scalable and high-availability AI workflows. *Int. J. Intell. Syst. Appl. Eng.* 12 (4), 5896–5905.
<https://doi.org/10.6084/m9.figshare.30445046>
17. Khaleq, A.A., Ra, I., 2021. Intelligent autoscaling of microservices in the cloud for real-time applications. *IEEE Access* 9, 35464–35476. <https://doi.org/10.1109/ACCESS.2021.3061890>
18. Lesniak, A., Laigner, R., Zhou, Y., 2021. Enforcing consistency in microservice architectures through event-based constraints. In: *Proc. 15th ACM Int. Conf. on Distributed and Event-Based Systems (DEBS)*, pp. 180–183. <https://doi.org/10.1145/3465480.3467839>
19. Li, C., Wang, C., Tang, H., Luo, Y., 2019. Scalable and dynamic replica consistency maintenance for edge-cloud system. *Future Gener. Comput. Syst.* 101, 590–604.
<https://doi.org/10.1016/j.future.2019.05.014>
20. Lodderstedt, T., Bradley, J., Labunets, A., Fett, D., 2025. Best current practice for OAuth 2.0 security. RFC 9700, Internet Engineering Task Force. <https://doi.org/10.17487/RFC9700>
21. Malaraju, S.K., Kakaraparathi, G.C., Bondalapati, R.C., 2025. Efficient deployment of scalable AI models via Kubernetes orchestration in hybrid cloud setups. In: *2025 Int. Conf. on Intelligent and Secure Engineering Solutions (CISES)*, pp. 1450–1454.
<https://doi.org/10.1109/cises66934.2025.11265182>
22. Meng, C., Song, S., Tong, H., Pan, M., Yu, Y., 2023. DeepScaler: Holistic autoscaling for microservices based on spatiotemporal GNN with adaptive graph learning. In: *38th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, pp. 53–65.
<https://doi.org/10.1109/ASE56229.2023.00038>
23. MongoDB, Inc., 2025. MongoDB Manual: Security (TLS/SSL, x.509, SCRAM) and read preference (maxStalenessSeconds). <https://www.mongodb.com/docs/manual/security/>
24. Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H., Kim, S., 2020. Horizontal pod autoscaling in Kubernetes for elastic container orchestration. *Sensors* 20 (16), 4621.
<https://doi.org/10.3390/s20164621>
25. Nunes, J.P.K.S., Nejati, S., Sabetzadeh, M., Nakagawa, E.Y., 2024. Self-adaptive, requirements-driven autoscaling of microservices. In: *Proc. 19th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 168–174.
<https://doi.org/10.1145/3643915.3644094>
26. Oracle, 2025a. Java Platform, Standard Edition Security Developer's Guide, Release 21. <https://docs.oracle.com/en/java/javase/21/security/>
27. Oracle, 2025b. MySQL 8.4 Reference Manual: Replication. <https://dev.mysql.com/doc/refman/8.4/en/replication.html>
28. PostgreSQL Global Development Group, 2025. PostgreSQL 17 Documentation: High availability, load balancing, and replication. <https://www.postgresql.org/docs/17/high-availability.html>
29. Rajendran, R.N., Anumula, S.K., Rai, D.K., Agrawal, S., 2025. Zero trust security model implementation in microservices architectures using identity federation. *arXiv preprint*. <https://doi.org/10.48550/ARXIV.2511.04925>
30. Reddy, D.N., Suryodai, R., Kumar, S.B.V., Ambika, M., Muniyandy, E., Krishna, V.R., Abdurasul, B., 2025. A scalable microservices architecture for real-time data processing in cloud-based applications. *Int. J. Adv. Comput. Sci. Appl.* 16 (9), 31–42.
<https://doi.org/10.14569/IJACSA.2025.0160905>
31. Sachidananda, V., Sivaraman, A., 2024. Erlang: Application-aware autoscaling for cloud microservices. In: *Proc. European Conf. on Computer Systems (EuroSys)*, pp. 888–923.
<https://doi.org/10.1145/3627703.3650084>
32. Schultz, W., Avitabile, T., Cabral, A., 2019. Tunable consistency in MongoDB. *Proc. VLDB Endow.* 12 (12), 2071–2081. <https://doi.org/10.14778/3352063.3352125>
33. Shafi, N., Abdullah, M., Iqbal, W., Erradi, A., Bukhari, F., 2024. CDAScaler: A cost-effective dynamic autoscaling approach for containerized microservices. *Clust. Comput.* 27 (4), 5195–5215.
<https://doi.org/10.1007/s10586-023-04228-y>
34. Spring Security Team, 2025. Spring Security Reference: OAuth 2.0 Resource Server. <https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/index.html>
35. Tan, J., Zhang, R., Zhang, T., Li, F., Chen, J., Zheng, Q., Zhang, P., Qiao, H., Shi, Y., Cao, W., 2019. iBTune: Individualized buffer tuning for large-scale cloud databases. *Proc. VLDB Endow.* 12 (10), 1221–1234. <https://doi.org/10.14778/3339490.3339503>

36. Wang, Z., Zhu, S., Li, J., Jiang, W., Ramakrishnan, K.K., Yan, M., Zhang, X., Liu, A.X., 2024. DeepScaling: Autoscaling microservices with stable CPU utilization for large scale production cloud systems. *IEEE/ACM Trans. Netw.* 32 (5), 3961–3976. <https://doi.org/10.1109/TNET.2024.3400953>
37. Xie, S., Wang, J., Li, B., Zhang, Z., Li, D., Hung, P.C.K., 2024. PBScaler: A bottleneck-aware autoscaling framework for microservice-based applications. *IEEE Trans. Serv. Comput.* 17 (2), 604–616. <https://doi.org/10.1109/TSC.2024.3376202>
38. Xu, M., Yang, L., Wang, Y., Gao, C., Wen, L., Xu, G., Zhang, L., Ye, K., Xu, C., 2024. Practice of Alibaba cloud on elastic resource provisioning for large-scale microservices cluster. *Softw. Pract. Exp.* 54 (1), 39–57. <https://doi.org/10.1002/spe.3271>
39. Zaccarini, M., Poltronieri, F., Borsatti, D., Cerroni, W., Foschini, L., Grabarnik, G.Y., Scotece, D., Shwartz, L., Stefanelli, C., Tortonesi, M., 2025. Chaos engineering based Kubernetes pod rescheduling through deep sets and reinforcement learning. In: *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS)*. <https://doi.org/10.1109/NOMS57970.2025.11073590>
40. ZargarAzad, M., Ashtiani, M., 2023. An auto-scaling approach for microservices in cloud computing environments. *J. Grid Comput.* 21 (4), 73. <https://doi.org/10.1007/s10723-023-09713-7>
41. Zhang, Y., Hua, W., Zhou, Z., Suh, G.E., Delimitrou, C., 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In: *Proc. 26th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp.167–181. <https://doi.org/10.1145/3445814.3446693>
42. Zhou, S., Mu, S., 2021. Fault-tolerant replication with pull-based consensus in MongoDB. In: *18th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pp.687–703. <https://www.usenix.org/conference/nsdi21/presentation/zhou>