International Journal of Science and Research (IJSR)
ISSN: 2319-7064
Impact Factor 2025: 7.089

A Multi-Criteria Fault-Tolerance Framework for Reliable Web Services in Service-Oriented Architectures

Anurag Tiwari[1], Shyam Srivastava[2], Pawan Kumar[3]

[1]Assistant Professor, Department of Business Management & Entrepreneurship, Dr. Ram Manohar Lohia Avadh University, Ayodhya, India
Email: anuragtiwari@rmlau.ac.in

[2, 3] Assistant Professor, Department of Business Management & Entrepreneurship, Dr. Ram Manohar Lohia Avadh University, Ayodhya, India

Abstract: ***Reliable service delivery is difficult in service-oriented and cloud-native systems because transactions cross independently deployed components, networks, platforms, and third-party services. This paper presents a revised three-segment fault-tolerance framework that combines runtime monitoring, recovery controls, and multi-criteria decision making. Segment 1 detects abnormal conditions and identifies functionally equivalent service candidates. Segment 2 evaluates candidates by availability, response time, failure rate, recovery time, consistency, and resource cost; Analytic Hierarchy Process (AHP) derives transparent criterion weights and Technique for Order Preference by Similarity to Ideal Solution (TOPSIS) ranks the alternatives. Segment 3 executes and verifies the selected failover or replication action and returns operational evidence to the monitoring loop. The framework extends an earlier conceptual design with recent research on microservice quality attributes, Kubernetes fault behavior, resilience patterns, automated robustness testing, and adaptive load balancing. It also specifies analytical measures, implementation considerations, and an empirical validation protocol for future evaluation.***

Keywords: fault tolerance, web services, service-oriented architecture, microservices, reliability, AHP, TOPSIS, quality of service

1. Introduction and Literature Review

Service-oriented architecture (SOA) organizes software capabilities as discoverable and interoperable services, enabling heterogeneous systems to collaborate through well-defined interfaces [1]. Cloud computing expanded this model by making computation, storage, and network resources elastic and remotely managed, but it also increased dependence on infrastructure and services outside the application's direct control [2]. Contemporary microservice systems intensify this distribution: an end-user transaction may require several service calls, and a localized timeout, resource leak, or network partition can propagate through the call chain. Consequently, service reliability cannot be reduced to the uptime of a single server.

1.1 Reliability and Architectural Quality

Reliability in a service ecosystem is closely related to availability, latency, recoverability, consistency, scalability, and resource efficiency. A systematic literature review by Li et al. [3] identified reliability and performance among the critical quality attributes of microservice architecture and showed that architectural tactics frequently create trade-offs across attributes. For example, aggressive replication can increase availability, but it also introduces coordination cost and can expose common-cause failures. Zhou et al. [4] demonstrated that reliability models for composite web services should account for such common-cause behavior instead of assuming that all component failures are independent. Architecture-based reliability analysis is therefore valuable because it relates component behavior and invocation structure to system-level outcomes [5].

1.2 Resilience Patterns and Cloud-Native Operations

Common resilience tactics include retries with bounded backoff, timeouts, circuit breakers, bulkheads, active or passive replication, health checks, failover, and service substitution. Their effectiveness depends on workload, failure mode, dependency topology, and parameter settings. Mendonça et al. [6] used formal models to analyze microservice resilience patterns and showed that patterns such as retry and circuit breaker should be configured as explicit design decisions rather than adopted as generic safeguards. Recent pattern research has also moved toward reusable development guidance: Miranda et al. [7] proposed a catalog of fault-tolerant microservice patterns derived from literature and practitioner input. Together, these studies support a policy-driven framework in which the fault response is selected from context rather than fixed permanently at design time.

Cloud-native orchestration supplies automatic restarts, placement, and health probes, but these mechanisms do not eliminate application-level failures. Flora et al. [8] found that Kubernetes probes can detect several failures, while some software-aging behaviors remain difficult for the platform to recognize and correct. This distinction matters because a container may remain technically alive even when its service quality has degraded. Therefore, fault detection should combine infrastructure signals with service-level indicators such as error rate, tail latency, stale responses, and recovery time.

1.3 Robustness Testing, Load Balancing, and Research Gap

Reliability claims also require systematic testing. Tsai et al. [9] emphasized the need to test and evaluate service-oriented software across functional and quality dimensions. More recently, Giamattei et al. [10] developed automated functional and robustness testing for microservice architectures using interface specifications and monitored executions. This work reinforces the value of fault injection, invalid-input testing, and causal analysis when validating recovery behavior. In parallel, Mushtaq et al. [11] showed that fault tolerance should be coordinated with task scheduling and load balancing because a recovery action can move work onto healthy but already overloaded resources. Zhang et al. [12] further demonstrated the practical value of combining adaptive load balancing, multi-level fault tolerance, and AHP-based QoS evaluation in a containerized web architecture.

These developments expose a persistent gap. Many fault-tolerance solutions specify the recovery mechanism but do not provide a transparent method for choosing among several functionally equivalent services when reliability, latency, recovery, consistency, and cost conflict. The earlier three-segment framework of Tiwari and Bharti [13] outlined monitoring, quality assessment, and reliability formulation, but its decision logic and empirical validation plan required further specification. The present paper extends that work by (a) clarifying the responsibilities and data flow of each segment, (b) integrating AHP and TOPSIS as a traceable selection mechanism, (c) connecting service-level reliability with modern cloud-native evidence, and (d) defining an evaluation protocol that avoids unsupported claims of effectiveness before experimentation.

2. Conceptual Foundation and Proposed Framework

2.1 Conceptual Foundation

A fault is the hypothesized or observed cause of an incorrect system state; an error is the part of the state that can lead to failure; and a failure occurs when delivered service deviates from its specification. In a web-service environment, crash faults stop a component, omission faults suppress expected messages, timing faults violate latency constraints, response faults return incorrect values, and network partitions isolate otherwise healthy nodes. Byzantine behavior is more difficult because replicas may return inconsistent or malicious results. Fault-scalable Byzantine protocols demonstrate that stronger guarantees are possible, but their coordination and resource costs must be considered [14].

The proposed framework separates fault detection from decision making and action execution. This separation supports auditability: the system can record which condition was observed, which criteria influenced the decision, which alternative was selected, and whether the result restored service. It also supports graceful degradation. When there are insufficient resources for active replication, the system can revert to a simpler point-to-point or passive failover policy while preserving the same monitoring and decision interfaces.

2.2 Proposed Three-Segment Framework

The framework operates as a closed feedback loop rather than a one-time service-selection procedure. Runtime observations enter through Segment 1, decision evidence is produced in Segment 2, and the chosen recovery action is executed and verified in Segment 3. The outcome then updates service histories and future selection scores.

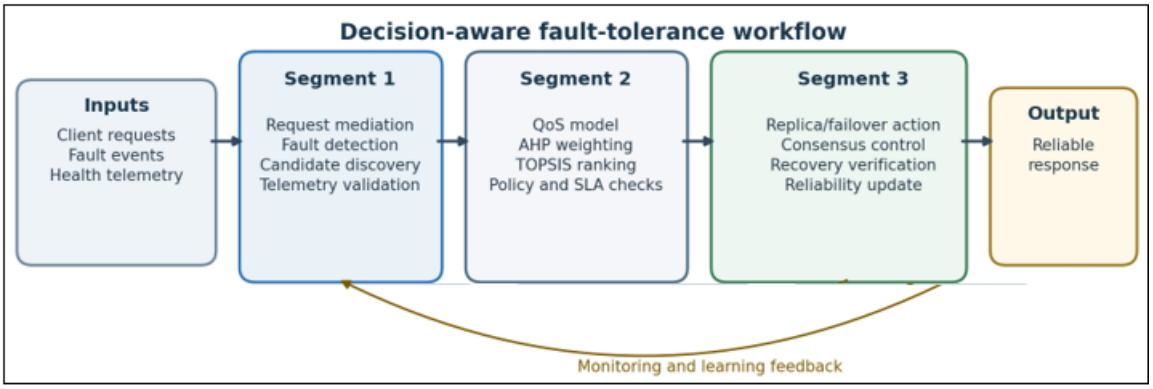

Figure 1: Decision-aware three-segment framework for reliable web services

2.2.1 Request Mediation and Fault Intelligence

Segment 1 is the operational interface between clients and the service ecosystem. It validates incoming requests, applies authentication and routing policy, and maintains a registry of candidate services that provide equivalent or substitutable functions. Its monitoring layer collects response codes, timeouts, latency percentiles, resource saturation, health-probe results, and semantic validation outcomes. A fault classifier converts these signals into a normalized incident record containing the affected service, suspected failure mode, severity, time window, and confidence.

The segment should avoid declaring a service unhealthy from a single noisy observation. Sliding windows, quorum-based health evidence, and debouncing rules can distinguish transient network variation from sustained degradation. Once a fault threshold is crossed, Segment 1 creates a candidate set from the registry and forwards current measurements and policy constraints to Segment 2.

2.2.2 QoS Assessment and Alternative Ranking

Segment 2 converts operational evidence into a service-selection decision. Each candidate is described by measurable criteria, while policy determines whether a criterion is a benefit (higher is preferred) or a cost (lower is preferred). AHP provides criterion weights that can reflect the needs of a particular application, such as prioritizing consistency for financial transactions or recovery speed for interactive services. TOPSIS then identifies the candidate closest to the ideal criterion profile and farthest from the least desirable profile.

Table 1: Suggested criteria for runtime service selection

Criterion	*Preference*	*Operational interpretation*
Availability	Benefit	Observed successful-service proportion or MTBF/MTTR estimate.
Reliability	Benefit	Probability of correct completion over the required mission interval.
Response time	Cost	Tail or percentile latency for comparable requests.
Failure rate	Cost	Weighted recent errors, timeouts, and invalid responses.
Recovery time	Cost	Expected time to restore acceptable service after a fault.
Consistency	Benefit	Degree to which the candidate satisfies required data semantics.
Resource cost	Cost	Compute, network, replication, and coordination overhead.

Note: Criteria may be added or removed according to the application's service-level objective; all measurements must use a common time window before ranking.

2.2.3 Recovery, Consensus, and Verification

Segment 3 translates the ranking into action. Depending on the failure model and policy, it may route the request to the highest-ranked replica, activate a passive standby, execute requests against several replicas, invoke compensation logic, or temporarily degrade a noncritical feature. When replicated services can return different values, the segment applies a consensus or adjudication policy appropriate to the trust and consistency requirements. A simple majority may be sufficient for benign response faults, whereas Byzantine settings require protocols with stronger assumptions and higher replica cost.

Recovery is not considered complete merely because an alternate endpoint responded. The segment validates response semantics, measures the achieved latency, records the final status, and updates the service history used by Segment 2. Failed recovery attempts trigger escalation to the next ranked alternative or a predefined safe-degradation policy.

3. Reliability and Multi-Criteria Formulation

3.1 Service-Level Measures

For a repairable service i, operational availability can be approximated from mean time between failures (MTBF) and mean time to repair (MTTR):

$$A_i = MTBF_i / (MTBF_i + MTTR_i). \quad (1)$$

If n replicas fail independently and any one correct replica is sufficient, the parallel reliability over mission time t is:

$$R_{parallel}(t) = 1 - \prod_{i=1}^{n} [1 - R_i(t)]. \quad (2)$$

The independence assumption is optimistic when replicas share software versions, databases, networks, credentials, or deployment zones. A common-cause factor should therefore be estimated from correlated incidents or represented explicitly in a dependency model [4]. The framework treats the analytical result as one input to decision making rather than a complete description of reliability.

3.2 AHP Weight Derivation

Let C = {c1, c2, ..., cm} be the decision criteria. Stakeholders compare each pair using Saaty's 1-to-9 scale, producing a reciprocal matrix P in which p_jk expresses the importance of criterion j relative to criterion k [15]. The normalized principal eigenvector gives the weight vector w, with each weight nonnegative and the weights summing to one. Judgment quality is checked through the consistency ratio:

$$CR = CI / RI, \text{ where } CI = (\lambda_{max} - m) / (m - 1). \quad (3)$$

A commonly accepted operational rule is to revisit the pairwise judgments when CR exceeds 0.10. Weights should be versioned with the application policy because a critical transaction and a background analytical request may require different priorities.

3.3 TOPSIS Ranking

Let x_{ij} be the measured value of candidate service i on criterion j. The decision matrix is normalized to obtain r_{ij}, and the weighted matrix is $v_{ij} = w_j r_{ij}$. The positive ideal vector contains the best value for each criterion, while the negative ideal vector contains the least desirable value. Euclidean distances D_i^+ and D_i^- are calculated from the two ideals, and the relative closeness score is [16]:

$$C_i = D_i^- / (D_i^+ + D_i^-), \text{ with } 0 \le C_i \le 1. \quad (4)$$

Candidates are ranked in descending order of C_i, subject to mandatory constraints such as security level, geographic residency, interface compatibility, and consistency semantics. This two-stage approach separates value judgments (AHP weights) from runtime measurements (TOPSIS scores), making the final decision explainable.

4. Operational Workflow and Implementation

4.1 Operational Decision Workflow

A decision cycle begins when Segment 1 detects a policy-relevant fault or degradation. The registry removes candidates that fail hard constraints. Segment 2 aligns all

remaining measurements to the same observation window, applies the current AHP weight profile, normalizes the decision matrix, and calculates TOPSIS closeness scores. Segment 3 attempts the highest ranked feasible action and validates the outcome. If the response fails semantic or timing checks, the next candidate is attempted. The cycle ends with an audit record containing measurements, weights, ranking, selected action, and verified result.

To prevent rapid oscillation between alternatives, deployments should include hysteresis, a minimum dwell time, and cooldown rules. Confidence bounds are also useful when candidates have few observations. Under severe resource pressure, the policy may lower replication degree, queue low-priority requests, or return a controlled partial response rather than cause a system-wide overload.

4.2 Observability and Data Quality

The ranking is only as reliable as its telemetry. Metrics must be time synchronized, tagged by service version and deployment zone, and separated by request class. Mean latency alone is insufficient because tail latency can dominate user experience. Traces should connect client requests with downstream calls, while logs should preserve failure type and recovery action. Missing or stale observations should reduce a candidate's confidence rather than be silently treated as good performance.

4.3 Replication and Consistency

Replication policies must match operation semantics. Read-only or idempotent operations can often be retried or raced across replicas safely. Non-idempotent operations require request identifiers, deduplication, transactional outboxes, or compensation to avoid duplicate effects. Strong consistency can reduce availability during partitions, whereas eventual consistency may be unacceptable for particular transactions. The consistency criterion in Table 1 therefore acts as a policy gate as well as a ranking attribute.

4.4 Fault Injection and Robustness Testing

Validation should include controlled crash faults, response delays, packet loss, malformed responses, stale data, CPU or memory saturation, and dependency failure. Automated interface-based tests can exercise both valid and invalid inputs, while distributed traces can identify where recovery paths fail [10]. Tests should also include software-aging scenarios because ordinary liveness probes may not capture gradual quality degradation [8].

5. Discussion, Evaluation, and Limitations

The principal advantage of the framework is explicit separation of observation, decision, and execution. This structure permits a service owner to change monitoring technology or recovery mechanism without discarding the decision model. It also makes trade-offs visible. For example, a low-latency replica may have a poor recovery history, while a highly available replica may incur additional coordination cost. AHP-TOPSIS does not remove these conflicts; it records how the policy resolves them.

The framework complements, rather than replaces, platform features. Kubernetes, service meshes, API gateways, and cloud load balancers can implement probes, retries, circuit breakers, and routing, but their actions should be coordinated with application-level semantics. Mushtaq et al. [11] show why fault tolerance, scheduling, and load balancing must be considered together, and Zhang et al. [12] provide evidence that adaptive load balancing and layered failover can improve the QoS of containerized web systems. The proposed decision layer offers a technology-neutral way to connect these controls to measurable objectives.

There are also costs. Pairwise comparisons become burdensome when too many criteria are used; historical measurements may lag behind sudden failures; TOPSIS rankings can be sensitive to normalization and correlated criteria; and replication can amplify traffic during an incident. For this reason, the framework should use a small, interpretable criterion set, apply hard safety constraints before ranking, and expose the final score breakdown to operators.

5.1 Evaluation Protocol

A future empirical evaluation should compare the proposed framework with at least three baselines: static round-robin routing, health-check failover, and fixed-weight QoS routing. Experiments should use a reproducible service graph and workloads representing read-only, idempotent, and state-changing operations. Fault campaigns should vary fault type, location, duration, and correlation. Each scenario should be repeated sufficiently to report confidence intervals rather than a single mean.

Primary outcomes should include successful-request rate, availability, p95 and p99 latency, mean time to detect, mean time to recover, incorrect-response rate, and additional compute and network cost. Secondary outcomes should include ranking stability, AHP consistency ratio, frequency of policy overrides, and the proportion of recoveries that require escalation to a second alternative. Sensitivity analysis should vary criterion weights and common-cause failure levels. This design would allow the framework's benefits and overheads to be evaluated without relying on theoretical proximity as a substitute for measured evidence.

5.2 Limitations and Future Research

The present contribution is a conceptual and analytical framework; it does not report a new production deployment or controlled experimental dataset. AHP judgments may reflect stakeholder bias, and standard TOPSIS assumes crisp measurements even when telemetry is uncertain. Future work should evaluate fuzzy or probabilistic extensions, learn context-specific weights from service-level outcomes, and study whether online ranking remains stable under rapidly changing loads. Additional research is needed for correlated and Byzantine faults, cross-region data semantics, security-aware service substitution, and energy-aware recovery.

6. Conclusion

Reliable web services require more than replication or a

single health check. They require a closed process that observes failures, evaluates feasible alternatives against multiple quality criteria, executes an appropriate recovery action, and verifies the result. The revised three-segment framework combines these responsibilities with an explainable AHP-TOPSIS selection method. By incorporating recent evidence on cloud-native fault behavior, resilience patterns, robustness testing, and coordinated load balancing, the framework provides a clearer foundation for future implementation and empirical study. Its value will ultimately depend on transparent policy, high-quality telemetry, and rigorous testing under realistic faults.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, Web Services: Concepts, Architectures and Applications. Berlin, Germany: Springer, 2004. doi: 10.1007/978-3-662-10876-5.

[2] M. Armbrust et al., "Above the Clouds: A Berkeley View of Cloud Computing," University of California, Berkeley, Technical Report UCB/EECS-2009-28, 2009.

[3] S. Li et al., "Understanding and Addressing Quality Attributes of Microservices Architecture: A Systematic Literature Review," Information and Software Technology, vol. 131, Art. no. 106449, 2021. doi: 10.1016/j.infsof.2020.106449.

[4] B. Zhou, K. Yin, S. Zhang, H. Jiang, and A. J. Kavs, "A Tree-Based Reliability Model for Composite Web Service with Common-Cause Failures," in Advances in Grid and Pervasive Computing, P. Bellavista et al., Eds. Berlin, Germany: Springer, 2010, pp. 418-429. doi: 10.1007/978-3-642-13067-0_44.

[5] S. S. Gokhale and K. S. Trivedi, "Reliability Prediction and Sensitivity Analysis Based on Software Architecture," in Proc. 13th Int. Symp. Software Reliability Engineering, 2002, pp. 64-75. doi: 10.1109/ISSRE.2002.1173214.

[6] N. C. Mendonça, C. M. Aderaldo, J. Cámara, and D. Garlan, "Model-Based Analysis of Microservice Resiliency Patterns," in Proc. IEEE Int. Conf. Software Architecture, 2020, pp. 114-124. doi: 10.1109/ICSA47634.2020.00019.

[7] F. de S. Miranda, D. S. dos Santos, R. F. Vilela, W. K. G. Assunção, R. C. dos Santos, and V. H. S. C. Pinto, "A Proposed Catalog of Development Patterns for Fault-Tolerant Microservices," in Proc. XXIII Brazilian Symp. Software Quality, 2024, pp. 406-416. doi: 10.1145/3701625.3701678.

[8] J. Flora, P. Gonçalves, M. Teixeira, and N. Antunes, "A Study on the Aging and Fault Tolerance of Microservices in Kubernetes," IEEE Access, vol. 10, pp. 132786-132799, 2022. doi: 10.1109/ACCESS.2022.3231191.

[9] W.-T. Tsai, X. Zhou, Y. Chen, and X. Bai, "On Testing and Evaluating Service-Oriented Software," Computer, vol. 41, no. 8, pp. 40-46, 2008. doi: 10.1109/MC.2008.380.

[10] L. Giamattei, A. Guerriero, R. Pietrantuono, and S. Russo, "Automated Functional and Robustness Testing of Microservice Architectures," Journal of Systems and Software, vol. 207, Art. no. 111857, 2024. doi: 10.1016/j.jss.2023.111857.

[11] S. U. Mushtaq, S. Sheikh, S. M. Idrees, and P. A. Malla, "In-Depth Analysis of Fault Tolerant Approaches Integrated with Load Balancing and Task Scheduling," Peer-to-Peer Networking and Applications, vol. 17, no. 6, pp. 4303-4337, 2024. doi: 10.1007/s12083-024-01798-5.

[12] P. Zhang, L. Xiang, Z. Song, and Y. Yang, "Adaptive Load Balancing and Fault-Tolerant Microservices Architecture for High-Availability Web Systems Using Docker and Spring Cloud," Discover Applied Sciences, vol. 7, Art. no. 705, 2025. doi: 10.1007/s42452-025-07320-7.

[13] A. Tiwari and A. K. Bharti, "Design of Fault-Tolerant Framework for Reliable Web Services," Journal of the Maharaja Sayajirao University of Baroda, vol. 55, no. 1(XIV), pp. 126-131, 2021.

[14] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-Scalable Byzantine Fault-Tolerant Services," in Proc. 20th ACM Symp. Operating Systems Principles, 2005, pp. 59-74. doi: 10.1145/1095810.1095817.

[15] T. L. Saaty, "Decision Making with the Analytic Hierarchy Process," International Journal of Services Sciences, vol. 1, no. 1, pp. 83-98, 2008. doi: 10.1504/IJSSCI.2008.017590.

[16] C.-L. Hwang and K. Yoon, Multiple Attribute Decision Making: Methods and Applications. Berlin, Germany: Springer, 1981. doi: 10.1007/978-3-642-48318-9.