



MICROSERVICES ARCHITECTURE OPTIMIZATION FRAMEWORK: A DYNAMIC SERVICE MESH AND AUTOMATED FAULT MITIGATION PROTOCOL FOR SCALABLE CLOUD SOFTWARE ENGINEERING

Mohamed Ahmed Elmubarak Mohamed ¹, Rania Elsheikh Hamid Ibrahim ², Abulfathi Ibrahim Saleh Al-Hussaini ^{3,*} and Hamid Basheer Almansor Adam ⁴

¹ Department of Computer Science, Faculty of Computer Science and Information Technology, Sudan Technological University - Nuba College - Information Systems Department, Sudan.

² Department of Management Information Systems, Sudan University of Science and Technology, Khartoum, Sudan.

³ Industry Faculty, College of Technology, American Open University Global, Ibadan, Nigeria.

⁴ Sudan Technological University, Sudan.

* Corresponding Author

International Journal of Science and Research Archive, 2026, 21(01), 118–129

Publication history: Received on 26 August 2026; revised on 02 October 2026; accepted on 05 October 2026

Article DOI: <https://doi.org/10.30574/ijrsra.2026.21.1.1855>

Copyright © 2026 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution License 4.0

ABSTRACT

Distributed microservices improve deployment independence and selective scaling, but remote calls introduce communication overhead and expose applications to dependency failures. This paper proposes the Microservices Reliability and Optimization Engine (MROE), a conceptual service mesh framework that combines a streamlined proxy data plane, telemetry-informed endpoint selection, and localized circuit breaking. The design uses bounded observation windows and exponentially weighted moving averages to summarize endpoint latency, error frequency, and queue utilization. A normalized endpoint cost guides request allocation, while a circuit-breaker state machine controls isolation and gradual re-entry. The framework separates local request handling from policy distribution and makes fallback behavior dependent on application semantics. We formalize sequential-path latency and request-based availability, describe a proposed routing and isolation procedure, and define a reproducible evaluation protocol covering load, injected failures, resource consumption, and component ablations. Hypothetical numerical scenarios illustrate metric calculations; they are neither experimental measurements nor validated performance predictions. A targeted review synthesizes published empirical findings on proxy overhead, adaptive control, and policy correctness while distinguishing them from MROE validation. The contribution is an explicit integration design and a testable evaluation plan. Implementation, calibration, and controlled benchmarking are required before claims about throughput, latency reduction, availability, or recovery time can be established.

Keywords: Microservices; Service Mesh; Adaptive Routing; Circuit Breaking; Fault Isolation; Cloud Software Engineering.

1 INTRODUCTION

Microservices organize application functionality into independently deployable units aligned with discrete business domains [1,2]. This architectural paradigm facilitates modular scalability and decoupled release pipelines. However, its

effectiveness is governed by domain decomposition boundaries, platform operational maturity, traffic profiles, and the inherent latency of distributed orchestration; adopting microservices does not intrinsically guarantee superior performance compared to a monolithic baseline.

Inter-service communication introduces cumulative overheads, including serialization, network transit, ingress/egress proxying, and queue dwell times. Under degraded conditions, downstream latency can saturate caller-side resources, driving systemic congestion across the dependency graph. Defensive patterns such as bounded resource pools and circuit breakers mitigate local starvation [3,4], while service meshes centralize telemetry aggregation and traffic shaping. Nevertheless, these mechanisms require harmonious tuning, as uncoordinated retries, out-of-date health signals, and simultaneous replica evictions can amplify cascading overload.

MROE is introduced as an architectural integration framework to coordinate proxy resource management, endpoint selection, and failure containment. Its primary design objective is to evaluate whether bounded, node-local telemetry can effectively direct request dispatching and fault isolation while upholding functional correctness and bounding computational overhead. The framework specifically targets deadline-driven, request-response workloads across replicated services. Persistent streams, un-replicated stateful dependencies, and multi-region synchronizations remain outside its operational envelope.

This paper formalizes an architectural decoupling of the local data path from the asynchronous policy-distribution plane, a parameterized routing and circuit-breaking specification, and an evaluation methodology configured to examine both steady-state throughput and degenerate failure modes. The contributions are conceptual; this work does not introduce an empirical software deployment, benchmark results, or assertions of absolute superiority over established service mesh implementations.

At distributed scale, central tendency metrics such as arithmetic mean latency frequently mask high-percentile tail spikes. Dean and Barroso [5] demonstrated that tail distributions dominate overall service responsiveness when queries fan out across numerous distributed components. This reality underpins MROE's emphasis on percentile tracking, deadline adherence, and end-to-end path accounting. In fan-out topologies, traffic balancing strategies that optimize solely for aggregate throughput can degrade user experience if they bias load toward nodes exhibiting volatile tail performance.

Robust observability serves as a fundamental prerequisite for dynamic, telemetry-driven traffic management. Dapper demonstrated that low-overhead, transparent distributed tracing can reconstruct transactions across service hops using adaptive sampling and standardized runtime instrumentation [6]. Accordingly, MROE treats trace propagation, clock synchronization fidelity, and sampling strategies as core evaluation parameters. Absent these controls, endpoint arbitration risks conflating network delay, queue dwell time, proxy latency, and local execution cost.

Fault containment controls must be calibrated against non-linear feedback loops. Contemporary research on metastable failures shows that reactive retries, sudden load re-allocation, and abrupt recovery mechanisms can entrench a system within sustained degraded states [7]. Standard reliability engineering practices therefore mandate strict retry budgets, dynamic load shedding, and graceful failure handling [8]. These findings guide MROE's integration of bounded request admission, strict retry limits, telemetry validity windows, and staged replica recovery states.

Finally, request-layer routing remains tightly coupled to underlying cluster capacity and container placement. Firmament established that centralized scheduling algorithms enhance task placement efficiency and response times at datacenter scale [9], while FIRM demonstrates the utility of proactive, SLO-driven resource arbitration for microservices [10,11]. Because MROE operates explicitly within the transient request-routing plane, it is designed to complement existing cluster-level resource schedulers rather than replace capacity-planning infrastructure.

2 MATERIALS AND METHODS

2.1 Review scope and architectural foundations

This targeted narrative review considers primary studies of mesh overhead, communication architecture, adaptive resource management, circuit breaking, and policy correctness. It compares the mechanisms and evaluation contexts most relevant to MROE rather than claiming a systematic or exhaustive review. Practitioner books and documentation provide architectural background; conference papers provide the principal empirical evidence. Findings from different testbeds are interpreted in their original settings, not combined into a numerical estimate of MROE performance.

Service decomposition creates deployable boundaries but also introduces remote-call and failure-handling obligations [1,2]. Circuit breakers, resource limits, and explicit recovery behavior are established reliability practices [3,4]. MROE

therefore treats these mechanisms as foundations. Its proposed contribution must lie in the specification and evaluation of their coordination, rather than in claiming their invention.

2.2 Proxy overhead and communication placement

Zhu et al. [12] introduce MeshInsight to decompose sidecar overhead. Their analysis separates inter-process communication, socket operations, protocol parsing, and filters, showing that bottlenecks vary with proxy configuration. This supports component-specific measurement rather than a single universal mesh penalty. Their compositional model does not establish tail-latency predictions under contention. MROE accordingly retains separate latency components and requires end-to-end measurement when testing local transport or filter changes.

Chen et al. [13] propose mRPC, which places RPC marshalling and policy enforcement in a managed system service. This is a direct architectural alternative to repeatedly parsing application-generated messages in sidecars. A lightweight sidecar is consequently only one possible response to communication cost. MROE preserves a proxy boundary for local observation and policy enforcement, but must justify its deployment and integration costs against alternatives such as managed RPC.

Saokar et al. [14] describe ServiceRouter, whose production architecture uses directly linked routing libraries for most RPC traffic and employs locality rings and shard-aware routing. This evidence emphasizes that endpoint equivalence and topology cannot be assumed for every workload. MROE is narrower: its base procedure addresses eligible replicas of one service and does not supply a general sharding or geo-distributed routing solution. The comparison motivates explicit membership and locality constraints.

2.3 Lifecycle costs and deployment alternatives

Cvetković and Klimovic [15] examine mesh behavior under serverless policy and sidecar churn. Their study distinguishes costs during active request processing from costs during initialization and configuration propagation. MROE adopts this distinction in its evaluation plan. A reduction in proxy initialization time would not alone establish faster application readiness, since scheduling, image access, application initialization, and policy distribution also contribute.

The deployment choice should depend on workload and operational requirements. Per-instance sidecars, shared proxies, routing libraries, and managed RPC services differ in lifecycle coupling, language integration, resource sharing, and enforcement boundaries. MROE is not proposed as a universal replacement for these designs. Its candidate benefit is a directly specified local decision procedure whose telemetry and actuation costs can be measured under matched functionality.

2.4 Adaptive resilience and fallback semantics

Saleh Sedghpour et al. [16] experimentally evaluate adaptive circuit-breaker control in an Istio and Kubernetes testbed. Their work already connects breaker settings to throughput and tail-response objectives. MROE therefore cannot claim that telemetry-driven circuit breaking is a previously unaddressed problem. Its hypothesis concerns the interaction of endpoint scoring, admission, recovery probing, and explicit valid-outcome accounting. A tuned adaptive breaker is an essential comparison, not merely an optional baseline.

Larsson et al. [17] investigate cached responses as a soft circuit-breaking actuator. Their experiments quantify a tradeoff between reduced traffic and stale data. This motivates application-specific fallback contracts, but cached reads cannot be assumed to satisfy transactional writes. MROE separates degraded responses from completed business operations and requires idempotency and deadline checks for retries. These requirements are design constraints; they have not yet been shown to improve MROE availability.

2.5 Telemetry management and configuration correctness

FIRM uses online observations and machine learning to localize contention and adjust resource provisioning [11]. It supplies evidence for adaptive management at the resource-allocation layer. MROE proposes routing and admission among existing eligible replicas, so its actuation scope differs. A deployment may need both allocation and routing control; either action can fail when the underlying service capacity is insufficient.

Zheng et al. [18] develop MeshTest to generate end-to-end tests for interactions between traffic-management configurations. Their findings motivate verifying policy behavior independently of performance measurements. Istio already exposes circuit-breaking and outlier-detection controls [19]. A meaningful MROE evaluation must therefore preserve equivalent security and protocol policies while testing whether coordinated decisions improve valid completion.

2.6 Synthesis and bounded contribution

Table 1 summarizes the comparison. The identified opportunity is an integration question: whether a transparent local rule can coordinate normalized endpoint observations, admission limits, and recovery states while maintaining application semantics. The reviewed work establishes relevant antecedents, including adaptive breaker control and soft fallbacks. The review does not establish that MROE is the first framework to combine these ideas. An implemented evaluation is required to determine whether its specific coordination provides a useful contribution.

Table 1: Relationship between selected prior approaches and MROE

Prior approach	Primary focus	MROE comparison requirement
MeshInsight Zhu et al. (2023)	Sidecar cost decomposition	Measure each optimization and contention effects
mRPC Chen et al. (2023)	Managed RPC and policy service	Justify proxy boundary and integration costs
ServiceRouter Saokar et al. (2023)	Library routing and locality	State replica, shard, and locality assumptions
Adaptive breaker Saleh Sedghpour et al. (2021)	Breaker control under overload	Compare against tuned adaptive control
Soft breaking Larsson et al. (2021)	Caching and staleness tradeoff	Define semantically valid fallbacks
FIRM Qiu et al. (2020)	Resource management from telemetry	Separate routing from capacity allocation
MeshTest Zheng et al. (2025)	Policy interaction correctness	Test outcomes and policies as well as latency

2.7 Scope and assumptions

The design assumes that eligible replicas expose equivalent functionality; request deadlines are propagated; observations are timestamped; endpoint selection respects locality and authorization rules; and queue or connection limits are available to the proxy. Local decisions use only admissible endpoints. A replica marked healthy is not assumed to have unlimited spare capacity. Stateful operations require affinity and consistency rules before rerouting is allowed.

Requests are selected for an endpoint before dispatch. An already dispatched operation is not transparently moved to another replica. A subsequent attempt is a retry and must satisfy the deadline, retry budget, and idempotency requirements. Fallback responses must expose their semantic status: a cached read, deferred action, and completed payment are different outcomes.

2.8 Latency model and objective

For a sequential execution path P with K services, the client-observed response time is decomposed as follows:

$$L_P = \sum_{i=1}^K (T_{proc,i} + T_{queue,i}) + \sum_{i=1}^{K-1} T_{net,i,i+1} + \sum_{j=1}^{J(P)} \delta_{proxy,j} + T_{edge} \quad (1)$$

$T_{proc,i}$ is processing time at service i ; $T_{queue,i}$ is its waiting time; $T_{net,i,i+1}$ is transport time between consecutive services; $J(P)$ counts the proxy traversals actually encountered; $\delta_{proxy,j}$ is the cost of one traversal; and T_{edge} includes ingress, client-boundary, and other explicitly measured edge costs. All terms use the same time unit. The $K-1$ transport links exclude client-boundary links. Measurements must avoid counting proxy or serialization time twice. Fan-out requires a critical-path model rather than summing parallel branches.

For a measurement interval W , request-based availability is defined using valid application outcomes completed within the deadline:

$$A_{req}(W) = \frac{\text{Nvalid and within deadline}(W)}{\text{Noffered}(W)} \quad (2)$$

Noffered includes all valid offered requests, including those rejected before reaching the application. This definition differs from time-based uptime. Degraded responses must be reported separately unless the service contract explicitly counts them as successful completion. Aggregate availability must also be broken down by transaction type so that failures in critical operations are not hidden by successful lightweight requests.

Let θ contain telemetry parameters, endpoint weights, admission limits, breaker parameters, and proxy settings. The design objective is to maximize A_{req} over feasible θ while satisfying a tail-latency SLO and resource budgets:

$$\text{maximize } A_{req}(\theta), L_{99}(\theta) \leq L_{SLO}, C(\theta) \leq C_{max}, M(\theta) \leq M_{max} \quad (3)$$

C and M denote CPU demand and memory consumption under a specified workload. Feasibility also requires functional correctness, security-policy equivalence, and bounded retry amplification. This is a design objective, not a solved optimization theorem; the proposed heuristic below must be evaluated against the stated constraints.

2.9 Architecture

The application layer contains the business services and defines deadlines, valid outcomes, idempotency rules, and fallback contracts. The proxy data plane enforces transport security and protocol policies, maintains bounded local observations, selects endpoints, and applies admission limits. The policy control plane distributes versioned configurations and endpoint membership. Local mitigation can continue with a valid cached policy when the control plane is temporarily unavailable.

Figure 1 shows the proposed relationships. Telemetry exported for analysis is separated from local decisions so that a remote collector is not required for every request. Configuration changes require validation, versioning, and rollback. A policy must identify both its expiration behavior and the safe action when fresh membership information is unavailable.

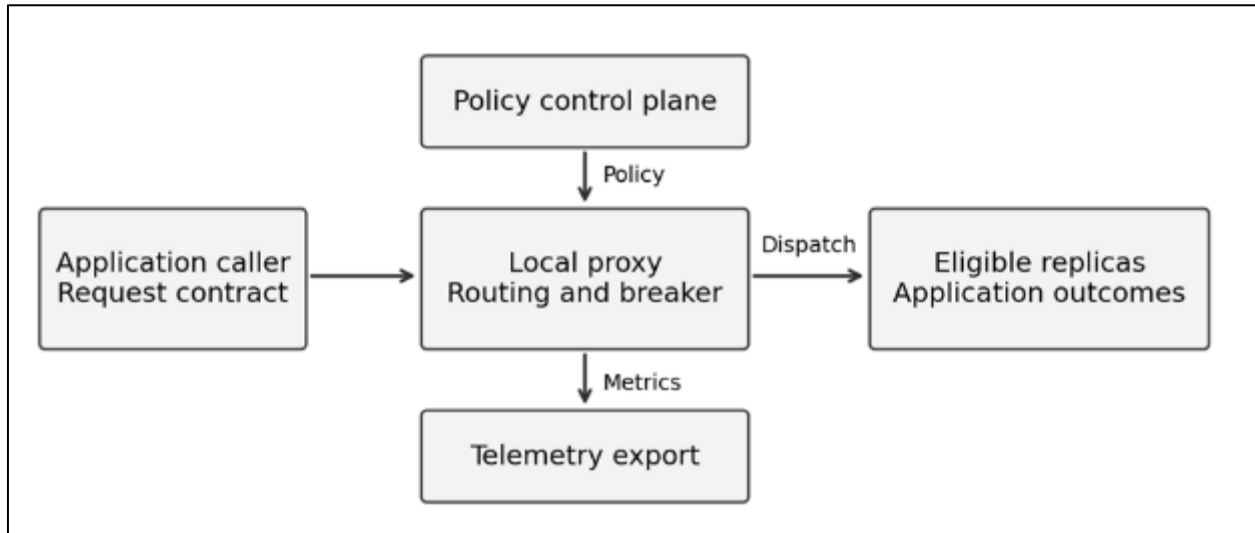


Figure 1: Proposed MROE architecture and local decision boundary

UDS is proposed only for communication between co-located components sharing an appropriate socket namespace and permissions. It does not replace transport between remote nodes. Persistent HTTP/2 connections are a candidate optimization where supported by the selected protocol. Filter removal must preserve authentication, authorization, encryption, observability requirements, and relevant protocol checks. Any performance comparison must preserve these capabilities across the mesh baselines.

2.10 Telemetry and endpoint cost

For endpoint e , observations are collected in bounded time windows of duration Δ . Each completed window records mean latency $\ell_{e,t}$, error fraction $f_{e,t}$, queue utilization $q_{e,t}$, sample count $n_{e,t}$, and observation time. Queue utilization is queue occupancy divided by a configured positive capacity. Errors include timeouts and connection failures, with their definitions fixed before evaluation. Latency samples alone must not hide timed-out attempts.

The proposed design smooths latency, error fraction, and queue utilization separately using an exponentially weighted moving average (EWMA):

$$\bar{x}_{e,t} = \alpha x_{e,t} + (1 - \alpha)\bar{x}_{e,t-1}, \quad 0 < \alpha \leq 1 \quad (4)$$

Here x denotes one of the three window statistics. The first sufficiently populated window initializes the estimate. An empty window does not imply zero latency or zero errors; its timestamp ages. Observations older than a configured freshness limit are treated as stale. The smoothing parameter, minimum sample count, freshness limit, and window duration are configurable and require calibration.

A proposed endpoint cost combines bounded normalized statistics:

$$S_{e,t} = w_L \min(\ell_{e,t}/\ell_{ref}, 1) + w_F f_{e,t} + w_Q \bar{q}_{e,t} \quad (5)$$

The nonnegative weights sum to one, and ℓ_{ref} is a positive reference latency specified per service. The normalized error and queue terms lie in $[0,1]$. Lower cost indicates a preferable endpoint. The bounded latency term prevents a scale mismatch with fractions; it is not used to hide deadline violations, which are tracked separately. Other normalization choices are possible and must be compared through sensitivity analysis.

An endpoint is admissible only when its breaker is closed, its observations satisfy freshness and sample requirements, its membership and security rules permit use, and its configured concurrency limit has not been reached. For this admissible set E , the proposed dispatch probability is:

$$p_{e,t} = \frac{\exp(-\beta S_{e,t})}{\sum_{u \in E} \exp(-\beta S_{u,t})}, \quad \beta > 0 \quad (6)$$

β controls concentration on lower-cost endpoints. Very large β may concentrate traffic enough to recreate congestion; very small β approaches equal allocation. New or stale endpoints use a separate bounded probe allowance to obtain observations rather than entering full allocation immediately. If E is empty, the request follows the application-defined fallback or rejection rule. No additional replica capacity is assumed.

2.11 Circuit breaker and request handling

The proposed breaker has closed, open, and half-open states. For a sufficiently populated fresh observation window, a trigger occurs when the error EWMA exceeds an error threshold or when both latency and queue EWMA exceed their respective thresholds. A configured number of consecutive triggering windows opens the breaker. Severe connection failures may use an explicit emergency rule, which must be documented and tested separately.

An open endpoint receives no normal traffic. After a configured cooldown, it becomes half-open and admits a bounded number of probes. Any failed probe reopens it. A configured number of consecutive successful probes below stricter recovery thresholds closes it. Recovery thresholds must be lower than trip thresholds to introduce hysteresis. Endpoint-local ejection does not authorize bypassing application consistency or security policies.

Algorithm 1 defines the proposed operational procedure. It is a specification for future implementation and has not been benchmarked.

- Load a validated policy and eligible endpoint membership; initialize bounded observations and breaker states.
- At each telemetry boundary, summarize completed attempts, timeouts, and queue occupancy; update populated EWMA and timestamps.
- For each endpoint, apply freshness checks, sample requirements, trigger persistence, and breaker state transitions.
- Before dispatch, verify the request deadline, transaction semantics, and caller admission budget.
- Form the admissible endpoint set using membership, policy, breaker state, freshness, and concurrency limits.
- If endpoints are admissible, calculate costs and probabilities; reserve one concurrency slot atomically and dispatch.
- If reservation fails, recompute admission within a bounded selection budget; otherwise apply fallback or reject.
- Admit half-open and new-endpoint probes only within their separate bounded budgets.
- Record each attempt and final application outcome, release reserved resources, and export aggregate observations.
- Permit a retry only when the remaining deadline, service-wide retry budget, and idempotency rules allow it.
- The algorithm is telemetry-driven and reactive to observed conditions. No predictive accuracy is claimed. A predictive extension would require a defined forecasting method and measurements of warning lead time, false alarms, missed failures, and decision overhead.

3 RESULTS AND DISCUSSION

This section summarizes experiments and engineering observations reported by other researchers. These are published findings about the systems named in Table 2, not experiments performed on MROE. The synthesis supports the plausibility of the proposed mechanisms and identifies evaluation priorities; it cannot verify MROE performance or transfer another system's improvement percentage to this framework.

Table 2: Selected published findings in their original evaluation contexts

Study and context	Finding reported by the source	Implication and transfer limit
Zhu et al. (2023) MeshInsight benchmark applications	Latency increased by 27–269%; CPU use by 42–163%, depending on configuration.	Profile protocol and configuration costs; these ranges are not universal mesh overhead.
Cvetković and Klimovic (2025) Serverless production-trace experiment	Applying mesh policies was associated with 38.3× higher P95 execution time and up to 3.9× higher P95 CPU utilization per worker.	Test policy and lifecycle churn separately; the serverless setting differs from steady microservices.
Qiu et al. (2020) FIRM across four microservice benchmarks	Up to 16× fewer SLO violations, up to 62% lower requested CPU limit, and up to 11× lower tail latency.	Adaptive resource control can help in tested settings; requested limits are not measured CPU consumption.
Chen et al. (2023) mRPC on DeathStarBench	Up to 2.5× speedup relative to a sidecar-based comparison.	Communication architecture matters; this is mRPC's reported gain, not a MROE prediction.
Saleh Sedghpour et al. (2021) Istio and Kubernetes overload tests	Adaptive control kept tail response below the target 98% of the time on average; availability was 70%, versus 63% for static control.	Report rejected requests as well as latency; availability definitions and overload conditions remain source-specific.
Larsson et al. (2021) Purpose-built service and Hipster Shop	One experiment reduced network load up to 30% with about 3% or less stale data; another cached about 80% of requests.	Evaluate staleness and semantics; read caching does not establish correct completion of writes.
Zheng et al. (2025) MeshTest on Istio and Linkerd	Found 23 new bugs, of which 19 were confirmed and 10 fixed at the time of reporting.	Functional policy tests are necessary; bug counts are not performance or availability improvements.

The reported maxima and ranges are preserved as source-specific results. They involve different workloads, baselines, metrics, and software implementations. A latency increase, a reduction in SLO violations, and the proportion of cached requests describe different quantities and cannot be averaged into one improvement claim. No pooled effect estimate or ranking of these systems is justified by this targeted synthesis.

Three implications follow for the proposed evaluation. First, configuration and protocol costs should be decomposed before choosing optimizations. Second, adaptive control must be assessed jointly with rejected, late, degraded, and valid outcomes so that apparent latency gains do not conceal lost functionality. Third, application and policy correctness should be checked alongside performance. These are interpretations of the reviewed evidence, not measured properties of MROE.

Published evidence does not establish the illustrative MROE values presented later. In particular, no cited study validates MROE's hypothetical throughput, failure fraction, or recovery interval. Establishing those quantities requires an implemented prototype, matched baselines, independent repetitions, and raw observation records as specified in Section 5.

3.1 Research questions and comparison design

The evaluation should determine whether MROE improves valid completion under faults, how routing affects latency under load, what resource overhead local telemetry introduces, and which components explain any change. The primary

comparison is against a tuned service mesh with equivalent protocol, security, retry, timeout, and resource settings. A default mesh configuration is a secondary reference. An unmeshed deployment and a monolith may provide architectural context, but differences in functionality and security must be disclosed.

All configurations should use the same application logic, input dataset, resource budget, and workload mix where equivalence is feasible. A monolithic implementation requires an explicit description of how service logic is combined and how state access and concurrency are matched. Experiments should randomize configuration order or block by testbed conditions to reduce temporal bias. Tuning data should be separated from the workloads used for final comparison.

3.2 Testbed and workload reporting

The testbed report must identify node count, CPU model and core allocation, memory, network capacity, operating system and kernel, Kubernetes and mesh versions, container images, placement, replicas, resource requests and limits, data stores, and autoscaling policies. The implementation report must identify proxy modifications, extension mechanisms, source commit, policy files, and the code implementing the proposed algorithm. No particular hardware or software version is assumed by this conceptual paper.

Select and identify the exact DeathStarBench application [20] or another documented workload. Supply its commit, dependency graph, dataset size, operation mix, and modifications. If payment and inventory operations are introduced, describe the extension rather than implying that every benchmark includes those services. An open-loop load generator can separate offered load from achieved completion rate, but generator capacity and client-side failures must be measured.

A candidate load sweep is 10,000 to 60,000 offered requests per second, adjusted to the capacity of the eventual testbed. These are proposed stress-test settings, not demonstrated system capacity. Report warm-up duration, measurement interval, random seeds, independent repetitions, and saturation criteria. Include steady demand, bursts, and transitions with autoscaling enabled and disabled.

3.3 Faults and metric definitions

Inject documented process crashes, added response delay, connection failures, and resource saturation. Candidate severities are 10%, 25%, and 50% of selected downstream replicas. Record affected services, replica selection, injection and restoration timestamps, fault duration, spare capacity, and workload intensity. Recovery with redundancy already available must be distinguished from recovery requiring new replicas.

Table 3: Operational definitions for the planned evaluation

Metric	Definition
Goodput	Valid application completions within the deadline per second
Latency	Client-observed end-to-end delay; report P50, P95, and P99 with timeout policy
Request availability	Valid timely completions divided by all valid offered requests
Failure rate	Failed or late final outcomes divided by offered requests; report degraded outcomes separately
Observed recovery time	Fault onset to sustained restoration of a predefined SLO; report required hold interval
CPU and memory	Absolute use for application, proxy, and control plane; normalize by completed work
Proxy overhead	Incremental proxy cost relative to an explicitly defined equivalent baseline
Startup time	Measured lifecycle boundary to proxy readiness; separate new initialization from reuse

Recovery time objective (RTO) is a target. An observed elapsed recovery time is a measurement and should be compared with that target. Detection, isolation, and sustained recovery times must be reported separately. The success condition for payment-like operations must require the intended business outcome; a fallback that declines or defers the operation cannot silently count as completed payment.

3.4 Statistical analysis and artifacts

Choose the number of independent runs using pilot variability and the precision required for the research questions. Report run-level estimates with 95% confidence intervals using a suitable method, such as resampling independent runs. Do not treat all requests within one run as independent experimental replications. Very high availability claims require sufficient observation duration, complete denominators, and uncertainty estimates that account for clustered failures.

Produce load-versus-goodput plots, latency distributions, fault timelines, and CPU and memory profiles. Conduct ablations for adaptive routing, fault isolation, connection reuse, eligible local UDS use, and proxy filter changes. Test telemetry windows, smoothing, β , threshold persistence, cooldowns, and probe budgets. Report adverse effects as well as gains, including oscillation, false ejection, telemetry cost, and loss of capacity from simultaneous isolation.

Archive source code, manifests, configuration versions, benchmark inputs, fault scripts, raw logs, and analysis scripts. Document commands needed to reproduce each figure and table and specify any access restrictions. Until those artifacts and measurements exist, the planned analysis cannot support an empirical performance claim.

All values in Tables 4 and 5 are hypothetical scenarios used to demonstrate calculations. They were not obtained from experiments, simulation, or an implemented MROE prototype. The architecture labels do not imply measured rankings, achievable capacity, or causal effects. No error bars or statistical tests are reported because there are no underlying observations.

Table 4: Hypothetical throughput and latency values for arithmetic illustration

Scenario label	Goodput requests/s	Mean latency ms	P99 latency ms
Monolithic reference	12400	85.40	310.00
Unmeshed reference	28500	32.10	145.20
Standard mesh reference	41200	21.80	88.50
MROE illustration	58900	14.20	38.60

For a higher-is-better metric, relative change is $100 \times (\text{candidate} - \text{reference}) / \text{reference}$. Using the hypothetical goodput values, $(58,900 - 41,200) / 41,200 \times 100 = 42.96\%$, or 43.0% to one decimal place. For a lower-is-better metric, reduction is $100 \times (\text{reference} - \text{candidate}) / \text{reference}$. The illustrative mean-latency reduction is 34.86%, and the P99 reduction is 56.38%. These calculations establish arithmetic relationships only.

Table 5: Hypothetical fault and lifecycle values for arithmetic illustration

Scenario or metric	Reference value	Candidate value	Relative reduction
Final failure fraction	42.80%	0.04%	99.91%
Observed recovery interval	184.50 s	1.12 s	99.39%
Proxy initialization interval	1,420 ms	180 ms	87.32%

An illustrative failure fraction of 0.04% implies 99.96% request availability under the same denominator if every outcome is either success or failure. It cannot simultaneously imply 99.995% availability under those same conditions. Different workloads or intervals could yield different values, but such distinctions would require measurements. Memory and CPU comparisons require absolute baseline values, a defined measurement scope, and observation records before a reduction percentage can be established.

The fault scenario also requires a capacity feasibility check. Losing half of the relevant replicas approximately halves their aggregate capacity if replicas are equivalent. Preserving throughput is possible only if sufficient spare capacity, reduced demand, valid alternative processing, or faster execution compensates for the loss. Routing alone cannot create missing service capacity. Any future near-zero failure claim must establish both this capacity condition and correct transaction outcomes.

3.5 Expected tradeoffs

The proposed routing cost may direct demand away from degraded replicas before caller queues fill, but it can also respond to noise or concentrate traffic on recently healthy endpoints. Smoothing introduces a tradeoff between responsiveness and stability. Queue pressure can reveal overload, while errors can reveal faults; their correlation does not guarantee that a single weighted score identifies the best action. Hysteresis, bounded probing, concurrency limits, and a service-wide retry budget therefore form part of the design rather than optional additions.

Local decisions avoid a collector round trip on the request path but may give different callers inconsistent views of the same endpoint. Simultaneous local ejections can eliminate usable capacity. The implementation should evaluate shared retry budgets, bounded ejection policies, and rate-limited recovery traffic without assuming that a globally synchronized health view is free or immediately available.

The selected endpoint probability can be computed once per telemetry update and cached. Updating an endpoint requires bounded local state, while rebuilding the allocation distribution requires work proportional to the number of replicas tracked for that service. Actual CPU, memory, synchronization, and contention costs depend on the implementation. No sub-second mitigation or resource savings follow from the architecture alone.

3.6 Validity and generalization

Construct validity depends on whether reported success means useful business completion, whether dropped and timed-out requests enter the denominator, and whether proxy overhead is distinguished from total CPU demand. Reporting only completed-request latency can understate user impact during failure; timeout counts and deadline violations must accompany latency distributions.

Internal validity depends on matched functionality, security, resource budgets, workload inputs, and tuning effort. Improvements produced by disabling required filters, changing application logic, or adding spare replicas cannot be attributed solely to routing. Ablations and tuned baselines are needed to separate these effects.

External validity is limited by the applications, protocol mix, deployment scale, service graph, and hardware tested. Results from a single benchmark would not establish effectiveness for stateful financial operations, long-lived streams, heterogeneous accelerators, or multi-region deployments. UDS benefits are local and cannot be extrapolated to remote network paths.

Statistical validity requires sufficient independent runs, explicit treatment of failure bursts, and appropriate uncertainty estimates. This conceptual paper supplies no measured observations, so it cannot resolve any of these validity questions empirically. Its numerical examples are explanatory and must not be used as operational forecasts.

4 CONCLUSION

This paper proposes MROE as a service mesh integration framework for adaptive endpoint allocation and telemetry-driven fault isolation. The design formalizes sequential-path latency, defines request availability in terms of valid timely outcomes, and specifies a local routing and circuit-breaker procedure with explicit admission, freshness, probing, and fallback rules. These definitions make the proposal testable and clarify the conditions under which rerouting is semantically and operationally feasible.

The illustrative scenarios demonstrate how relative changes and complementary success and failure fractions should be calculated. They do not establish throughput, availability, latency, recovery, or resource improvements. The next research step is an implementation followed by controlled comparisons, ablations, calibration, and reproducible artifact release. Predictive control and multi-region extensions should be considered only after the basic local mechanism and its correctness have been evaluated.

The broader literature also clarifies the conditions under which the framework could be useful. Tail-aware measurement is needed to distinguish a genuine improvement in user-visible responsiveness from a change that only improves the mean. Distributed tracing is needed to attribute delay to the correct service, queue, transport, or proxy component. Overload research shows that the safety of an adaptive policy depends on its feedback dynamics, including retry amplification, load shedding, and recovery pacing ([8]; Dean & Barroso, 2013; [7]; [6]).

The proposed contribution is consequently best understood as a testable coordination model. It brings together endpoint scoring, admission control, circuit-breaker state, freshness-aware probing, and application-defined fallback semantics in one local request procedure. The design does not remove the need for autoscaling, cluster scheduling,

tracing infrastructure, or service-specific reliability engineering. Those systems provide complementary control and measurement capabilities; MROE supplies a precise place to evaluate their interaction at the service-routing boundary.

A complete follow-up study should implement the protocol, publish its configuration and workload artifacts, compare it with a tuned adaptive breaker and representative mesh baselines, and report confidence intervals for tail latency, valid completion, recovery time, and resource cost. The study should also include negative results and failure cases, because a policy that protects one SLO by harming another is not a general optimization. Until those tests are completed, the numerical scenarios in this paper remain illustrative and the framework should be treated as a research specification rather than a validated production claim.

STATEMENTS ON COMPLIANCE WITH ETHICAL STANDARDS

Acknowledgments

The authors express their gratitude to their affiliated institutions for academic and institutional support. Funding: This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Disclosure of conflict of interest

The authors declare no conflict of interest.

Data Availability

This article presents a conceptual framework. The numerical examples are hypothetical and no experimental dataset is reported.

Code Availability

No implemented MROE prototype or executable research artifact is presented with this manuscript.

REFERENCES

- [1] Lewis, J., Fowler, M. (2014, March 25). Microservices: A definition of this new architectural term. MartinFowler.com. <https://martinfowler.com/articles/microservices.html>
- [2] Newman, S. 2021. Building microservices: Designing fine-grained systems (2nd ed.). O'Reilly Media.
- [3] Montesi, F., Weber, J. 2016. Circuit breakers, discovery, and API gateways in microservices. arXiv. <https://doi.org/10.48550/arXiv.1609.05830>
- [4] Nygard, M. T. 2018. Release it! Design and deploy production-ready software (2nd ed.). Pragmatic Bookshelf.
- [5] Dean, J., Barroso, L. A. 2013. The tail at scale. Communications of the ACM, 56(2), 74-80. <https://doi.org/10.1145/2404781.2404800>
- [6] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C. 2010. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report. <https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/>
- [7] Huang, L., Magnusson, M., Muralikrishna, A. B., Estyak, S., Isaacs, R., Aghayev, A., Zhu, T., Charapko, A. 2022. Metastable failures in the wild. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22) (pp. 73-90). USENIX Association. <https://www.usenix.org/conference/osdi22/presentation/huang-lexiang>
- [8] Beyer, B., Jones, C., Petoff, J., Murphy, N. R. (Eds.). 2016. Site reliability engineering: How Google runs production systems. O'Reilly Media. <https://sre.google/sre-book/table-of-contents/>
- [9] Gog, I., Schwarzkopf, M., Gleave, A., Watson, R. N. M., Hand, S. 2016. Firmament: Fast, centralized cluster scheduling at scale. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (pp. 99-115). USENIX Association. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog>
- [10] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J. 2016. Borg, Omega, and Kubernetes. ACM Queue, 14(1), 70-93. <https://doi.org/10.1145/2898442.2898444>
- [11] Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., Iyer, R. K. 2020. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In 14th USENIX Symposium on Operating Systems

- Design and Implementation (OSDI 20) (pp. 805-825). USENIX Association. <https://www.usenix.org/conference/osdi20/presentation/qiu>
- [12] Zhu, X., She, G., Xue, B., Zhang, Y., Zhang, Y., Zou, X. K., Duan, X., He, P., Krishnamurthy, A., Lentz, M., Zhuo, D., Mahajan, R. 2023. Dissecting overheads of service mesh sidecars. In Proceedings of the 2023 ACM Symposium on Cloud Computing (pp. 142-157). Association for Computing Machinery. <https://doi.org/10.1145/3620678.3624652>
- [13] Chen, J., Wu, Y., Lin, S., Xu, Y., Kong, X., Anderson, T., Lentz, M., Yang, X., Zhuo, D. 2023. Remote procedure call as a managed system service. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI (pp. 141-159). USENIX Association. <https://www.usenix.org/conference/nsdi23/presentation/chen-jingrong>
- [14] Saokar, H., Demetriou, S., Magerko, N., Kontorovich, M., Kirstein, J., Leibold, M., Skarlatos, D., Khandelwal, H., Tang, C. 2023. ServiceRouter: Hyperscale and minimal cost service mesh at Meta. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23) (pp. 969-985). USENIX Association. <https://www.usenix.org/conference/osdi23/presentation/saokar>
- [15] Cvetković, L., Klimovic, A. 2025. Towards a lightweight sidecar-based service mesh for serverless. In Proceedings of the 16th ACM Symposium on Cloud Computing (pp. 860-866). Association for Computing Machinery. <https://doi.org/10.1145/3772052.3772210>
- [16] Saleh Sedghpour, M. R., Klein, C., Tordsson, J. 2021. Service mesh circuit breaker: From panic button to performance management tool. In Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems (pp. 4-10). Association for Computing Machinery. <https://doi.org/10.1145/3447851.3458740>
- [17] Larsson, L., Tärneberg, W., Klein, C., Kihl, M., Elmroth, E. 2021. Towards soft circuit breaking in service meshes via application-agnostic caching. arXiv. <https://doi.org/10.48550/arXiv.2104.02463>
- [18] Zheng, N., Qiao, T., Liu, X., Jin, X. 2025. MeshTest: End-to-end testing for service mesh traffic management. In 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25) (pp. 301-316). USENIX Association. <https://www.usenix.org/conference/nsdi25/presentation/zheng-naiqian>
- [19] Istio Authors. n.d.. Circuit breaking. <https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/>
- [20] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., . . . Delimitrou, C. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud, edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 797-812). Association for Computing Machinery. <https://doi.org/10.1145/3297858.3304013>