Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Risk-Aware Degraded-Mode Orchestration for Resilient Cloud–Edge AI Agents: From Containerized Fault Injection to Heterogeneous Kubernetes Validation

Albert Adusei Brobbey[1]✉[2]

albertab@primemeridianintl.com

Prime Meridian International

Narayan Bhosale[1]

Crestmont University

Dan Bamfo[3]

Prime Meridian International

Research Article

Keywords: cloud–edge computing, AI agents, resilience orchestration, graceful degradation, fault injection, safety

Posted Date: August 10th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10499935/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Abstract

Cloud–edge AI agents depend on model, retrieval, policy, and tool services whose partial failure can turn ordinary availability mechanisms into unsafe continuation. We introduce DMO-AI, a risk-aware degraded-mode orchestrator that selects normal, rate-limited, cached, local, read-only, handoff, or blocked execution from task risk and observed dependency health. We first evaluate DMO-AI in a single-host containerized testbed with real HTTP chains and Toxiproxy faults: two frozen stages cover 141,000 executions, conventional baselines, ablations, sensing, and overhead. We then perform a complete 8,000-event validation across an x86-64 cloud node and ARM64 edge node running K3s, Istio/Envoy, real SmolLM2 inference, and a transactional SQLite tool. In the multi-node study, DMO-AI with Istio achieved 93.95% safe completion versus 64.70% for Istio alone, a paired 29.25-percentage-point gain (95% bootstrap CI 27.30–31.85), with no observed policy violations, duplicate commits, negative inventory, or high-risk edge substitutions. Mean latency increased from 139.67 to 158.83 ms. The results support explicit risk-bounded degradation as a semantic control layer that complements transport resilience without claiming production-wide generality.

1 Introduction

Cloud-native AI agents are distributed service-chain workloads. A single task may invoke a model endpoint, retrieve external context, consult a policy service, and execute a tool before returning a result. Kubernetes, service meshes, timeouts, retries, circuit breakers, and tracing improve the availability of such chains [1–9], but they do not decide whether an agent remains authorized to act when evidence or control services are missing. This distinction matters at the cloud edge, where partial connectivity is expected and local continuation can preserve service while also exceeding the action boundary intended by the operator.

Generic resilience policies fail in two directions. Fail-open behavior can preserve nominal completion by allowing ungrounded output, stale policy use, or partial tool execution. Fail-closed behavior prevents those outcomes but can make low-risk work unnecessarily unavailable. Retry-only and circuit-breaker policies improve transport reliability, yet remain largely task-agnostic: the same network state can require different responses for a low-risk summary and a high-risk write action. Agent frameworks therefore need a control plane that joins service health, task risk, evidence freshness, and action authority [10–15].

Consider three tasks arriving while the policy service is unreachable. A low-risk drafting task may remain useful if it continues with an explicitly time-bounded cached policy. A medium-risk analysis may continue only if the evidence cache is fresh enough for its risk class and the result is marked uncertain. A high-risk task that would execute a payment, alter a record, or trigger an operational tool should not inherit the same fallback merely because the transport failure is identical. The required response is therefore a function of both infrastructure state and task semantics. Conventional availability controls observe the former but ordinarily have no representation of the latter.

This distinction also separates service completion from useful and authorized completion. A request can return HTTP 200 while omitting required retrieval, relying on expired authority evidence, or executing only part of a state-changing tool chain. Conversely, a blocked high-risk action can be a correct safety outcome even though it lowers raw completion. For this reason, our primary outcome is safe completion: the fraction of tasks that complete without a specified unsafe condition or policy violation. The metric makes failure handling visible instead of rewarding a controller for merely returning a response.

We propose degraded-mode orchestration for AI (DMO-AI). The framework makes degraded behavior explicit and auditable: a task may continue normally, use a bounded cache, switch to local inference, become read-only, hand off, or block. This paper advances the prior design from a synthetic simulation to a measured container testbed. The testbed exercises real HTTP calls through Toxiproxy, uses a controller-blind safety oracle, applies identical task and fault sequences to all controllers, compares five conventional baselines, and isolates six DMO-AI components through two frozen stages.

The contributions are fivefold. First, we formalize degraded-mode selection as constrained runtime control rather than completion maximization. Second, we provide an implementable architecture and risk-conditioned policy matrix. Third, we report a prospectively frozen paired fault-injection confirmation comprising 33,000 executions and a separate 108,000-execution sensing-and-overhead extension. Fourth, we validate the mechanism in a genuine heterogeneous K3s deployment with a named Istio/Envoy data plane, real cloud and edge model servers, and a transactional tool across 8,000 additional paired events. Fifth, we separate confirmatory evidence from pilot-driven and infrastructure-driven refinement and release the workload, event logs, analysis code, deployment manifests, and environment evidence as a reproducibility package.

The evaluation addresses four questions: RQ1, whether DMO-AI improves safe completion relative to conventional resilience controllers; RQ2, which risk-aware fallback components account for that result; RQ3, whether active dependency sensing is necessary under the frozen fault schedule; and RQ4, what decision, audit, network, and policy-state overhead the reference implementation introduces. RQ5 asks whether the safe-completion advantage persists when execution crosses heterogeneous cloud–edge nodes and the comparator uses a real service mesh, lightweight model inference, and transactional state.

The scope is deliberately narrower than a claim of production-ready autonomic control. We evaluate controller behavior under deterministic, repeatable service impairments and a single two-node cloud–edge topology. The first two stages use an emulated model and transport-policy comparator; the third uses real SmolLM2 servers and Istio/Envoy but generates only one token because it tests routing and control rather than answer quality. These boundaries permit controlled comparisons while leaving multi-site replication, model-quality evaluation, asynchronous human response, and domain-specific safety cases for subsequent validation.

The remainder of the paper proceeds as follows. Section 2 positions DMO-AI against cloud-native resilience, cloud–edge orchestration, agent reliability, and runtime assurance. Section 3 defines the task,

health, authority, feasibility, and cost models. Section 4 presents the architecture, invariants, policy modes, and runtime loop. Section 5 describes the container testbed, multi-node deployment, frozen protocols, controllers, oracle, and statistical analysis. Section 6 reports confirmatory, ablation, stratified, sensing, overhead, and Kubernetes validation results. Sections 7 and 8 discuss interpretation, deployment, and validity, before Sections 9 and 10 address reproducibility and conclusions.

2 Background and Related Work

2.1 Cloud-native resilience

Container orchestration and microservices decompose applications into independently deployable services [1, 2]. Distributed tracing links requests across those services [3], while site-reliability engineering uses service-level objectives, health checks, load shedding, redundancy, and incident analysis to control failure [4–6]. Retries, timeouts, bulkheads, and circuit breakers suppress cascading faults [7]; chaos engineering tests whether those controls behave as intended [8, 21]. Zero-trust architecture adds continuous verification and least privilege [9]. These mechanisms remain necessary in an agent stack, but their unit of control is typically a request or service rather than the semantic risk and authority of the task. Cloud–edge orchestrators such as MiCADO-Edge and cloneMAP demonstrate policy-driven runtime management and fault-tolerant multi-agent deployment [18, 19]. Service-mesh experiments nevertheless show that sidecars carry measurable latency and resource costs, especially on constrained edge nodes [20, 22].

2.2 Agent service chains and failure semantics

Large-language-model agents combine planning, memory or retrieval, and tool use [10–12]. Retrieval-augmented generation makes output dependent on an external evidence service [13]. A policy or authorization service may further determine whether an action is permissible, and a tool gateway turns text generation into an external state change. Consequently, an HTTP success code does not establish a safe task outcome. Missing retrieval can yield ungrounded output; stale policy can invalidate authority; and a partially available tool chain can leave external state inconsistent. ReAct, Reflexion, and Toolformer illustrate how reasoning becomes coupled to environmental actions and API calls [23–25]. Production-stress, tool-risk, and adversarial-safety benchmarks increasingly evaluate agents under API faults, high-stakes tool use, or untrusted tool content [26–29].

2.3 From graceful degradation to risk-aware degradation

Graceful degradation preserves a subset of system value when full service is unavailable. For AI agents, the subset must be selected by risk. NIST's AI Risk Management Framework and the OWASP guidance for language-model applications emphasize validity, security, oversight, and constrained tool use [14, 15]. DMO-AI translates these concerns into runtime modes. Recent work in Cluster Computing also shows the value of criteria-driven choices and formal multi-objective models across the cloud–edge continuum [16, 33]. Runtime-assurance architectures similarly separate an advanced controller from a safety-

preserving fallback [30–32]. Our focus is complementary: selecting an agent's permitted behavior after deployment as service evidence and action authority change. Recent cloud-native agent architectures identify stratified autonomy, fault-tolerant components, and observability as deployment requirements [34], while classical AI-safety work motivates explicit constraints against unsafe side effects [35].

2.4 Positioning and research gap

Four research streams meet in this problem but do not individually resolve it. Cloud-native resilience provides mechanisms for detecting and containing service failure, yet typically optimizes request availability. Cloud–edge orchestration chooses placement and resources across heterogeneous infrastructure, but usually treats an application task as schedulable work rather than an authorization-bearing action. Agent-safety benchmarks expose tool-use and environmental risks, but generally evaluate model or policy behavior rather than runtime service degradation. Runtime-assurance architectures provide a principled separation between advanced and safety-preserving control, but they do not specify how retrieval, policy, tool, and model dependencies should map to agent-specific fallback modes.

Table 1
Positioning of DMO-AI against adjacent research streams

Research stream	Primary control concern	Unresolved issue for agent degradation
SRE and resilience [4–8, 21]	Health, latency, failure containment	Usually request-centric; no task-risk or authority semantics
Zero trust [9]	Continuous verification and least privilege	Does not select useful degraded task behavior
Cloud–edge orchestration [16, 18, 19, 33]	Placement, scaling, energy, heterogeneous resources	Limited treatment of evidence freshness and agent action authority
Service-mesh studies [20, 22]	Transport policy and sidecar overhead	Cannot infer semantic safety of a fallback
Agent architectures [10–13, 23–25, 34]	Planning, retrieval, memory, and tool use	Failure-mode governance is not the primary control abstraction
Agent safety benchmarks [26–29]	Tool risk, prompt injection, production stress	Limited degraded-service orchestration and availability comparison
Runtime assurance [30–32, 35]	Safety-preserving fallback and constrained control	No dependency-specific mode matrix for cloud–edge agents
DMO-AI	Health + risk + freshness + authority	Evaluated in controlled single-host and one two-node cloud–edge topology

DMO-AI contributes a connective control abstraction: the permitted degraded mode. It does not replace Kubernetes, a service mesh, an authorization service, a model gateway, or a safety monitor. Instead, it

consumes their observations and policies to decide what subset of agent behavior remains permissible for the present task. The novelty lies in joining infrastructure health with task-level authority and in evaluating that join against both availability-oriented baselines and component ablations.

3 Problem Formulation

3.1 Task, dependency, and health model

Let a task be x = (r, D, a, τ), where r ∈ {low, medium, high} is risk, D is the required service set, a denotes whether the task may change external state, and τ is the age of cached evidence. Let h map each service to {healthy, degraded, outage}. The controller selects m from M = {normal, bounded retry, rate-limited, local model, retrieval cache, cached policy, read-only, human handoff, block}. For a portfolio T, binary variable z_{tm} equals one when task t is assigned to mode m.

The dependency set is drawn from S = {model, retrieval, policy, tool}. The model service produces or transforms content; retrieval supplies task-specific evidence; policy evaluates scope and action authority; and the tool service changes or queries external state. These roles are not interchangeable. A model outage may permit bounded local substitution, whereas a policy outage can remove the evidence that a write remains authorized. Similarly, a tool impairment may allow advice to continue in read-only mode even when the requested state change cannot safely occur.

We distinguish latent health h* from observed health $\hat{h}$. The benchmark controls h* through fault injection, while the controller receives only $\hat{h}$ from dependency probes. In production h* is unknowable and $\hat{h}$ may be delayed, noisy, or incomplete. This separation prevents the experiment from granting the controller oracle access. It also makes sensing a testable mechanism: a policy can be logically sound yet ineffective when its observations do not reveal when a fallback is needed.

Risk r is an exogenous task classification, not a model confidence score. It summarizes the consequence of an incorrect, unauthorized, stale, or incomplete action. The write indicator a marks tasks capable of altering external state. Evidence age τ is interpreted relative to a risk-conditioned freshness threshold θr. In the reference policy θlow = 300 seconds, θmedium = 120 seconds, and θhigh = 30 seconds. These values are experimental policy parameters rather than universal recommendations.

3.2 Feasibility and constrained objective

For each task, $F_t = M(r_t, h_{D_t}, \tau_t, a_t)$ is the feasible set after applying risk, evidence-freshness, write-authority, retry, and fallback constraints. Let u_{tm} be unsafe-completion probability or a calibrated surrogate, ℓ_{tm} expected latency, q_{tm} cascade-risk contribution, and c_{tm} normalized operating and opportunity cost. With non-negative policy weights α + β + γ + δ = 1 and latency reference Lref, the portfolio objective is:

$$\min \sum_{t \in T} \sum_{m \in M} z_{tm} \left[\alpha u_{tm} + \beta(\ell_{tm}/L_{ref}) + \gamma q_{tm} + \delta c_{tm}\right] \quad (1)$$

The assignment is subject to exactly one mode per task and mode feasibility:

$\sum_{m} z_{tm} = 1;\ z_{tm} = 0 \text{ for } m \notin F_t;\ z_{tm} \in \{0,1\}$ (2)

Portfolio constraints require minimum safe completion κ, maximum policy-violation rate ε, and bounded use B_k of each constrained fallback resource k:

$|T|^{-1}\sum_{t}\sum_{m} z_{tm} s_{tm} \geq \kappa;\ |T|^{-1}\sum_{t}\sum_{m} z_{tm} v_{tm} \leq \varepsilon;\ \sum_{t}\sum_{m} z_{tm} b_{tmk} \leq B_k$ (3)

Here s_{tm} indicates safe completion under mode m, v_{tm} indicates a policy violation, and b_{tmk} represents consumption of a bounded fallback resource such as human review capacity, local compute, retry budget, or tolerated cache age. The first two constraints express service-level safety objectives; the third prevents a controller from preserving availability by silently exhausting a scarce fallback. A production optimizer could estimate these quantities from monitoring and domain risk models. The present study instead evaluates a transparent deterministic policy whose choices can be audited directly.

The formulation is a constrained binary assignment problem, but the present runtime does not claim to solve it globally. Weights, thresholds, and budgets are deployment policy and cannot be inferred universally from this testbed. DMO-AI approximates the problem lexicographically: remove infeasible modes; reject risk or authority violations; preserve safe completion with the least restrictive permitted substitute; and use handoff or blocking when no autonomous mode is feasible.

3.3 Cost and opportunity model

For deployment-specific economic calibration, the normalized cost term decomposes compute C, network N, risk-weighted staleness S, human labor H, denied-task opportunity O, and fallback-quality loss Q. Non-negative weights are selected by the operator and the component values are normalized to a common reference interval; energy-aware cloud–edge models offer one basis for calibrating the compute component [33]:

$c_{tm} = w_C C_{tm} + w_N N_{tm} + w_S S_{tm} + w_H H_{tm} + w_O O_{tm} + w_Q Q_{tm}$ (4)

Table 2
Degraded-mode cost components

Mode	Direct cost	Risk or opportunity cost
Normal	Cloud inference and ordinary service-chain cost	Base latency and compute
Bounded retry	Additional requests and backoff	Latency and cascade pressure
Rate-limited	Reduced request admission	Delay or partial-utility loss
Local model	Edge compute and energy	Reduced-quality opportunity cost
Retrieval cache	Avoided live retrieval	Risk-weighted staleness increasing with τ
Cached policy	Temporary continuity	Authority-window risk increasing with τ and a
Read-only	Suppressed state-changing tool	Partial-utility loss
Human handoff	Labor and response delay	Human-capacity budget
Block	No execution cost	Opportunity cost of unserved task

The decomposition clarifies why a single scalar such as latency is insufficient. Local inference can reduce network delay while increasing edge energy and reducing output quality. Human handoff can minimize autonomous risk while introducing labor cost and queue delay. Blocking has little direct compute cost but can impose a large opportunity cost. Cached policy and retrieval modes trade current network dependence for staleness exposure. Keeping these terms explicit allows different deployments to retain the same feasibility rules while calibrating economic weights to their own objectives.

3.4 Safety invariants and runtime complexity

The reference policy enforces four invariants. I1, no high-risk task may use the local-model fallback. I2, no state-changing tool action may complete when tool or policy authority is unavailable unless the mode suppresses the write or transfers control to a human. I3, cached evidence may be used only within the threshold of the task's risk class. I4, when no mode satisfies the first three invariants, the controller must hand off or block. These invariants define feasibility; they are not soft terms that can be outweighed by a latency or completion benefit.

Given a fixed mode set M and required dependency set D, the rule-based selector checks each dependency and at most each candidate mode once. Its decision complexity is $O(|D| + |M|)$, which is constant for the nine modes and four service roles used here. The dominant runtime cost is therefore health sensing and downstream execution rather than policy selection. This matters at the edge, where a sophisticated global optimizer may be inappropriate on a per-task critical path even if it remains useful offline for selecting thresholds and budgets.

The implemented selector is a deterministic policy realization rather than a learned optimizer. High-risk tasks cannot use local-model fallback. Retrieval and policy caches have risk-specific age limits of 300, 120, and 30 seconds for low, medium, and high risk. A missing tool service suppresses tool execution, while a high-risk task missing policy or tool authority is handed off or blocked. This explicit rule set makes the decision reproducible and inspectable.

Algorithm 1

Risk-bounded degraded-mode selection

Input: task $x=(r,D,a,\tau)$, observed health h, policy limits P

$F \leftarrow$ modes satisfying dependency, freshness, risk, write-authority, and budget constraints

if normal $\in F$ and every required dependency is healthy: return normal

remove any mode that can complete a write without current authority

if high risk and policy or tool authority is unavailable: return handoff if feasible, else block

if a required tool is impaired: return read-only

if policy is impaired: return cached policy only when fresh for r, else block

if retrieval is impaired: return fresh retrieval cache, else handoff/block as permitted

if model is impaired: return local model only for low/medium risk, else handoff/block

return rate-limited; after execution emit the structured audit event

The algorithm returns the first least-restrictive mode that satisfies the invariants under the observed state. Because the health observation can differ from latent truth, policy correctness does not guarantee outcome correctness; sensor error and stale observations remain part of the system risk. This is why the extension evaluates active sensing separately rather than treating health as a perfect input.

4 DMO-AI Design

4.1 Architecture

DMO-AI has five logical components: a health sensor, task-risk classifier, policy matrix, mode selector, and audit emitter. Health detection is separated from action permission. The sensor reports dependency state; the classifier describes the task; the matrix defines allowed substitutes; the selector chooses the least restrictive safe mode; and the emitter records state, mode, outcome, latency, and reason for escalation or blocking. Figure 1 shows the control path.

The health sensor probes only dependencies required by the current task. This avoids treating an unrelated outage as a reason to degrade every task and limits sensing traffic for short service chains. Each probe returns a categorical state derived from reachability and latency. The experimental

implementation uses a 30-ms threshold to distinguish healthy from degraded responses and records the aggregate sensing latency and bytes. A production implementation would normally add hysteresis, moving windows, and out-of-band telemetry to avoid reacting to a single noisy observation.

The risk classifier and dependency profile form the semantic input. In the testbed these attributes are generated with the task and are therefore deterministic from the controller's perspective. In a deployed system they could originate from workflow metadata, a tool registry, an authorization manifest, or a domain-specific classifier. Importantly, DMO-AI does not infer permission from the language model's output. Authority remains an external policy fact, and the mode selector may only narrow—not expand—the actions permitted by that fact.

The policy matrix is the inspectable bridge between semantics and infrastructure. Its rows represent risk and action conditions; its columns represent dependency states and candidate modes. The matrix carries cache-age limits, local-model eligibility, write suppression, handoff availability, and blocking rules. Because these controls are explicit, a reviewer can reconstruct why a task continued, changed behavior, or stopped without interpreting opaque model reasoning.

4.2 Modes and policy constraints

Table 3
DMO-AI modes and constraints

Mode	Trigger	Permitted behavior
Normal	Required services healthy	Full workflow
Bounded retry	Transient degradation; no authority loss	Limited attempts with backoff
Rate-limited	Residual pressure with authority intact	Reduced admission or delayed work
Local model	Cloud model impaired; low/medium risk	Emulated local inference; no high-risk use
Retrieval cache	Retrieval impaired; cache fresh for risk	Cached evidence with uncertainty flag
Cached policy	Policy impaired; cache fresh for risk	Time-bounded authority evidence
Read-only	Tool impaired or write authority incomplete	Advice/output; no state-changing tool
Human handoff	High-risk task lacks critical evidence	Acknowledged escalation; no autonomous action
Block	No safe feasible substitute	No autonomous completion

Normal and rate-limited modes preserve the live service chain and differ mainly in admission pressure. Local-model and retrieval-cache modes replace a missing computational or evidentiary dependency.

Cached-policy mode is more restrictive: it treats prior authorization evidence as temporarily usable only within a risk-specific window. Read-only mode preserves informational utility while suppressing state change. Human handoff and block are terminal autonomous modes; the former records an escalation acknowledgement, while the latter records that no feasible continuation exists.

4.3 Dependency-specific failure semantics

Table 4
Dependency-specific failure semantics and prohibited outcomes

Condition	Lost capability	Permitted response	Prohibited outcome
Model degraded/outage	Generation unavailable or too slow	Local model for low/medium risk; handoff for high risk	High-risk local substitution
Retrieval outage	Required evidence unavailable	Fresh cache; otherwise handoff or block	Ungrounded medium/high-risk completion
Policy outage	Current authority cannot be verified	Fresh cached policy within risk window; otherwise block/handoff	Write without valid authority
Tool degraded/outage	External action unavailable or uncertain	Read-only output or handoff	Partial or unauthorized state change
Multiple impairments	Several preconditions unavailable	Compose only compatible restrictions; otherwise block	Fallback interaction bypassing an invariant
Telemetry failure	Audit delivery unavailable	Task decision remains governed by policy; alert asynchronously	Treating audit failure as permission expansion

Multiple impairments require composition, not independent fallback selection. For example, a model outage may permit local inference, but a simultaneous policy outage still prevents an unauthorized write. The implemented selector therefore orders authority-preserving conditions before availability-preserving substitutions. Tool impairment triggers read-only behavior before model substitution, and high-risk loss of critical policy or tool authority triggers handoff or blocking before any cache or local path can complete the action.

4.4 Orchestration loop

For every task the orchestrator (i) senses required services, (ii) reads the task risk and dependency profile, (iii) checks evidence and cache freshness, (iv) selects a permitted mode, (v) executes within the mode's retry and action budget, and (vi) emits an audit event. The decision is per task, not per deployment, because service health can be identical while risk and authority differ. A runtime middleware, sidecar, or service-mesh extension can host this loop; transport controls remain in the mesh while semantic risk stays in the agent runtime.

The general mode set includes bounded retry, but the frozen DMO-AI controller deliberately did not select that mode. Retry behavior was isolated in the retry-budget, circuit-breaker, and service-mesh comparators so that DMO-AI could not receive credit for transport recovery in addition to its semantic fallbacks.

Mode selection is reevaluated per task rather than held as a global deployment state. This design avoids a coarse failure mode in which one high-risk outage forces every low-risk request to block, or one permissive low-risk fallback remains active when a high-risk write arrives. It also supports auditability: each event binds the task attributes, health vector, selected mode, completion result, and safety label at a single decision point.

4.5 Deployment integration

DMO-AI can be placed in agent middleware, a policy-enforcement sidecar, or an application-aware extension adjacent to a service mesh. The health and routing plane remains responsible for reachability, timeouts, load balancing, and transport retries. The DMO-AI layer consumes those signals together with task metadata and decides whether the agent may invoke the resulting route. This separation keeps semantic authority close to the workflow while reusing mature infrastructure for network control.

A production deployment would also require signed or otherwise integrity-protected policy metadata, versioned fallback rules, monotonic cache timestamps, bounded queues for human review, and a fail-safe response when the risk classifier is unavailable. Audit events should be correlated with traces but should not become a synchronous authorization dependency unless the domain explicitly requires durable pre-action logging. The reference implementation uses synchronous audit delivery only to measure its direct cost.

5 Experimental Method

5.1 Testbed

The primary evaluation uses a single-host containerized cloud–edge testbed. Four Python 3.12 HTTP services represent model, retrieval, policy, and tool dependencies. Toxiproxy 2.12.0 [17] sits on every path and injects latency or disables a proxy. Docker Engine 29.5.2 and Docker Compose 5.3.1 isolate the services and benchmark process. Calls to health and functional endpoints are real HTTP operations. The local-model mode performs bounded local computation, and human handoff is an emulated 10-ms acknowledgement; neither is claimed as a production model or measured human response.

Each dependency exposes a health endpoint and a role-specific functional endpoint: model invocation, retrieval, policy authorization, or tool execution. The benchmark container sends JSON over HTTP through a dedicated Toxiproxy listener for each dependency. Toxiproxy modifies the network path rather than the service implementation, allowing the same service code to be observed under healthy, delayed,

and unreachable conditions. The benchmark records sensing, dependency, decision, CPU, network-byte, retry, and resident-memory measures for every controller–task execution.

Table 5
Testbed components and evidence boundaries

Component	Implementation	Role in the evaluation
Host	Single Apple ARM64 host	Controls the evidence boundary; no multi-node claim
Container runtime	Docker Engine 29.5.2; Compose 5.3.1	Isolates benchmark, four dependencies, and Toxiproxy
Services	Python 3.12 HTTP processes	Model, retrieval, policy, and tool roles
Fault injector	Toxiproxy 2.12.0	35-ms latency with 8-ms jitter or disabled proxy
Local fallback	Bounded in-process computation	Emulation; no model-quality claim
Human handoff	10-ms acknowledgement	Emulation; no human-response claim
Extension telemetry	Fifth deterministic HTTP service	Measures one structured audit emission per task

5.2 Workload and fault schedule

Each run generates 300 tasks from a seeded distribution: 55% low, 30% medium, and 15% high risk in expectation. Risk-dependent Bernoulli draws determine retrieval, policy, tool, and write requirements; cache age follows a gamma distribution. A deterministic task-indexed schedule creates seven 8%-of-run fault windows: degraded model, retrieval outage, policy outage, degraded tool, simultaneous degradation of all services, simultaneous policy and tool outage, and model outage. Degradation adds 35 ms latency with 8 ms jitter; outage disables the proxy. The remaining intervals are healthy. Every controller receives the same task objects and latent fault schedule within a seed.

Across the 3,000 unique confirmatory task instances seen by DMO-AI, retrieval was required by 53.1%, policy by 58.8%, and a tool by 43.4%; 21.5% could change external state. These requirements overlap, so a single task may traverse several dependencies. Risk affects both consequence and service demand: high-risk tasks are more likely to require retrieval, policy, tools, and write authority than low-risk tasks. The workload therefore tests not only whether a controller detects a fault, but whether it interprets the loss of a dependency in the context of the task that needs it.

Table 6
Deterministic fault schedule within every frozen run

Task-index interval	Injected state	Operational interpretation
0–8%, 16–20%, 28–32%, 40–44%, 52–56%, 64–68%, 76–80%, 88–100%	Healthy	Reference operation and recovery intervals
8–16%	Model degraded	35-ms latency plus jitter
20–28%	Retrieval outage	Retrieval proxy disabled
32–40%	Policy outage	Policy proxy disabled
44–52%	Tool degraded	35-ms latency plus jitter
56–64%	All services degraded	Latency injected on all four paths
68–76%	Policy and tool outage	Both proxies disabled
80–88%	Model outage	Model proxy disabled

The ordered windows deliberately include recovery periods between most faults, making state transitions observable and preventing one outage from contaminating the entire run. The same order is also a limitation: it cannot reveal sensitivity to every temporal ordering or correlated failure process. We retain it for the frozen comparison and treat randomization of window order and duration as external validation rather than modifying the protocol after seeing results.

5.3 Controllers and ablations

Table 7
Compared controllers

Controller	Decision rule
Fail-open	Continues normal execution and permits partial retrieval/policy evidence
Fail-closed	Blocks when any required service is not healthy
Retry budget	Retries impaired dependencies with a three-attempt cap
Circuit breaker	Blocks on outage; one bounded retry on degradation
Service mesh	Transport-aware retry/block policy without task-risk semantics
DMO-AI	Full risk, cache, local, read-only, handoff, and block policy
Six ablations	Remove risk semantics, local fallback, retrieval cache, policy cache, handoff, or active health sensing

The service-mesh comparator implements a common transport-aware retry/block policy in the shared harness so that every controller receives identical tasks and latent faults. It is not a benchmark of a named Istio or Linkerd release; the comparison therefore isolates policy semantics rather than sidecar implementation performance.

The baselines span the conventional design space rather than representing tuned commercial products. Fail-open maximizes continuation and exposes the cost of permissive partial execution. Fail-closed provides a safety-oriented availability lower bound. Retry budget, circuit breaker, and service-mesh policy add increasingly structured transport recovery without task semantics. The five primary DMO-AI ablations remove one mechanism at a time while preserving the remaining rules. A sixth ablation in the extension removes active sensing while preserving the policy logic, separating the value of the rules from the value of knowing when to apply them.

5.4 Controller-blind oracle and outcomes

Safety labels are produced after execution by an oracle that receives the task, injected latent state, selected mode, completion status, and missing services. It does not inspect controller identity. A policy violation occurs when a write completes without valid policy or tool authority, or through an expired policy cache. Unsafe outcome includes any policy violation, high-risk local-model execution, expired retrieval evidence, or missing required retrieval for medium/high-risk tasks. Safe completion requires completion without either label. The primary outcome is safe completion; primary safety outcomes are unsafe and policy-violation rates.

The oracle evaluates realized execution rather than the controller's stated intention. A cached retrieval is unsafe when its age exceeds the risk-conditioned threshold. A write is a policy violation when current policy or tool authority is absent and the selected mode neither suppresses the write nor transfers control. Missing required retrieval is unsafe for medium- and high-risk tasks; for low-risk tasks it is recorded as uncertainty rather than automatically labeled unsafe. All violations are also unsafe outcomes, but not every unsafe outcome is an authorization violation.

Blocking and handoff are not labeled unsafe merely because autonomous completion does not occur. This prevents the oracle from rewarding availability at the expense of an explicit safety rule. Safe completion is therefore stricter than raw completion and more informative than the unsafe rate alone: a controller can have zero observed unsafe outcomes by blocking nearly everything, which is why completion, blocking, and latency are reported alongside safety.

5.5 Confirmatory design and analysis

A five-run pilot exposed unsafe use of stale retrieval and high-risk local fallback. Before confirmatory analysis, we tightened cache freshness, prohibited high-risk local inference, validated the change on a separate seed, and froze the benchmark and analysis hashes. Pilot and validation observations are excluded. The confirmatory study uses ten seeds (292001–292010), 300 tasks per controller, eleven controllers, and 33,000 executions. Run-level means and percentile 95% confidence intervals use 5,000

paired bootstrap resamples. Primary comparisons are DMO-AI versus the five conventional baselines. No controller logic was changed after the freeze.

Measured secondary outcomes include completion, blocking, wall-clock mean and p95 latency, decision overhead, process CPU time, observed network bytes, retries, and maximum resident set size. Because controllers execute sequentially on one host, latency and resource results are supporting evidence rather than a distributed scalability claim. Safety and completion comparisons retain task- and fault-level pairing.

The unit of independent resampling is the seeded run, not the individual event. For each metric, the analysis computes a run-level controller mean and resamples the ten paired runs 5,000 times with replacement. Paired differences are calculated within seed before aggregation, preserving the common task and fault schedule. Percentile intervals are descriptive uncertainty intervals for this frozen design; they are not substitutes for replication across hardware, domains, or fault generators.

Primary inference is restricted to DMO-AI versus the five conventional controllers for safe completion. Ablation and operational metrics explain mechanism and cost. Risk-stratified, fault-window, and mode-distribution results reported later are explicitly descriptive post hoc analyses of the frozen event logs. They are included to expose where the aggregate effect arises, not to introduce additional confirmatory claims.

5.6 Prespecified sensing and overhead extension

A second protocol was frozen before its outcomes were examined and leaves the primary confirmation unchanged. It adds a fifth deterministic HTTP telemetry service, one structured audit emission per task, and a no-health-sensing ablation that retains the DMO-AI rules but performs no dependency probes and receives an all-healthy observation vector. Ten new seeds (294001–294010), 900 tasks per controller, and twelve controllers produce 108,000 additional executions. The primary extension contrast is DMO-AI minus DMO-AI without health sensing for safe completion, again using 5,000 paired run-level bootstrap resamples.

The extension records audit-request latency and request/response bytes, service-level sensor false-positive and false-negative rates against the latent injected state, and the deep in-process size of the explicit policy matrix plus current observed health state. Sensor error rates are undefined for the no-sensing ablation and are not interpreted from its stored zero values. Byte counts cover serialized JSON request and response bodies and exclude transport headers. These diagnostics measure the benchmark implementation, not total service-mesh memory or a full OpenTelemetry pipeline.

For offline validation, the benchmark audit record includes the injected latent-fault count alongside the observed-fault count. Latent state is never passed to a controller or used to choose a mode; it is benchmark-only ground truth that would not be available to a production deployment.

5.7 Reproducibility controls

The primary benchmark source hash, extension source hash, seeds, controller list, task counts, outcomes, and analysis plan are stored in machine-readable freeze records. Pilot runs use seeds 290001–290005, the primary confirmation uses 292001–292010, and the extension uses 294001–294010. Their directories are separate. The extension validator checks seed membership, controller counts, task uniqueness, total rows, audit delivery, aggregate run counts, paired comparisons, and source hashes. The release manifest records byte counts and SHA-256 hashes for the raw and derived artifacts.

5.8 Heterogeneous Kubernetes validation

To test whether the mechanism survives the strongest evidence boundaries of the container study, we executed a third paired evaluation on two real K3s 1.36.2 nodes connected by a private WireGuard underlay [36, 37]. The cloud control-plane node was an x86-64 AWS EC2 t3.large instance with two virtual CPUs and 7.6 GiB memory in us-east-1. The edge worker was an ARM64 Ubuntu 24.04 virtual machine with five virtual CPUs and 4.8 GiB memory on the local edge host. The captured WireGuard metadata records a live handshake and 145.8 MiB received/401.8 MiB sent at the cloud peer. Workload placement labels kept the cloud model, policy, and transactional tool on the cloud node and the fallback model on the edge node.

Istio 1.30.3 injected native Envoy sidecars into the model gateways, policy service, tool service, and benchmark driver [38]. Model backends were deliberately unmeshed so that the gateway represented the controlled service boundary. Both sites ran the Q4_K_M GGUF build of HuggingFaceTB/SmolLM2-135M-Instruct through llama.cpp [39]. Inference was deterministic and limited to one generated token: the validation asks whether a real inference path is reachable and returns non-empty output, not whether the label is linguistically or semantically high quality. The state-changing workflow reserved inventory through a transactional SQLite service with idempotency keys, before/after state hashes, and a reset before each controller–seed pair [40].

Table 8
Heterogeneous cloud–edge validation environment

Element	Implementation	Experimental role
Cloud node	AWS EC2 t3.large; x86-64; 2 vCPU; 7.6 GiB	K3s control plane; cloud model, policy, tool
Edge node	Ubuntu 24.04 ARM64 VM; 5 vCPU; 4.8 GiB	K3s worker; fallback model
Network	WireGuard 10.88.0.0/24	Encrypted cloud–edge underlay
Service mesh	Istio/Envoy 1.30.3 sidecar mode	Named transport baseline and telemetry
Models	SmolLM2-135M-Instruct Q4_K_M via llama.cpp	Real cloud and edge inference; one-token output
Tool	Transactional SQLite inventory reservation	Real mutation, idempotency, integrity hashes

Four controllers received identical task objects and ordered fault windows within each seed: direct availability-first execution, Istio transport control, DMO-AI semantic control without mesh-specific retry, and DMO-AI composed with Istio. The two availability-first controllers explicitly failed open when live policy was unavailable; this configuration is a stress baseline, not a claim that Istio itself requires fail-open behavior. Each of 20 seeds (296001–296020) supplied 100 tasks to each controller, yielding 8,000 events. The schedule allocated 25% healthy operation, 15% cloud-model outage, 15% cloud-model degradation, 15% policy outage, 15% tool outage, and 15% compound policy/tool outage. A 1.2-s pause at window boundaries exceeded the one-second base ejection time.

The primary retained contrast was DMO-AI with Istio minus Istio for seed-level safe completion. Secondary contrasts were DMO-AI minus direct, DMO-AI with Istio minus DMO-AI, and Istio minus direct. We used the seed as the independent resampling unit and 5,000 paired percentile-bootstrap resamples. Safety audits additionally checked negative inventory, duplicate commits, commits under expired cached authority, high-risk edge substitution, non-empty successful model output, and agreement between committed events and persisted reservations.

An earlier infrastructure attempt used the same 20 seeds but 300 tasks per controller and eight-token generation. It created no retained result package: after more than six hours the local edge node became unavailable, Kubernetes removed the benchmark pod, and its temporary volume was lost. No aggregate outcomes from that attempt were computed or inspected. Before the retained restart, we reduced tasks to 100 per controller and generation to one token, pinned the benchmark driver to the cloud node, wrote every event durably, and used a host-persistent result directory. The seeds, controllers, fault proportions, outcomes, and paired analysis were unchanged. Only the complete 8,000-row restart is analyzed.

6 Results

6.1 Primary outcomes

DMO-AI achieved 97.53% [97.00%, 98.10%] safe completion, with 0.00% [0.00%, 0.00%] unsafe outcomes and 0.00% [0.00%, 0.00%] policy violations. The paired safe-completion gain was 18.47 pp [17.60, 19.40] over fail-open, 39.10 pp [38.27, 39.97] over fail-closed, 20.10 pp [19.30, 20.87] over retry budget, 20.13 pp [19.30, 20.97] over circuit breaker, and 20.10 pp [19.27, 20.90] over service mesh. All five safe-completion intervals exclude zero.

Table 9
Confirmatory outcomes across ten paired runs (95% bootstrap confidence intervals)

Controller	Safe completion	Unsafe	Violation	Blocked	Mean ms	p95 ms
Fail-open	79.07% [78.53%, 79.57%]	9.67% [9.00%, 10.33%]	7.53% [6.97%, 8.23%]	0.00% [0.00%, 0.00%]	45.90 [44.84, 47.10]	199.02 [187.85, 214.32]
Fail-closed	58.43% [57.77%, 59.10%]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	41.57% [40.90%, 42.23%]	26.41 [25.98, 26.82]	91.96 [87.69, 98.33]
Retry budget	77.43% [77.00%, 77.90%]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	48.90 [47.01, 51.20]	202.55 [186.89, 222.79]
Circuit breaker	77.40% [76.97%, 77.87%]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	22.60% [22.13%, 23.03%]	44.18 [42.84, 45.70]	199.01 [187.52, 213.83]
Service mesh	77.43% [77.00%, 77.90%]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	3.27% [2.77%, 3.80%]	45.72 [44.83, 46.65]	198.38 [184.64, 217.83]
DMO-AI	97.53% [97.00%, 98.10%]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	1.13% [0.77%, 1.50%]	36.31 [34.54, 38.78]	143.04 [133.14, 156.44]

6.2 Safety–availability balance

Fail-open completed 88.73% [88.20%, 89.23%] of tasks but only 79.07% [78.53%, 79.57%] safely; its unsafe and policy-violation rates were 9.67% [9.00%, 10.33%] and 7.53% [6.97%, 8.23%]. Fail-closed eliminated those observed outcomes but safe completion fell to 58.43% [57.77%, 59.10%] and blocking rose to 41.57% [40.90%, 42.23%]. DMO-AI occupied a different region of the trade space: 97.53% [97.00%, 98.10%] safe completion with only 1.13% [0.77%, 1.50%] blocking.

Retry budget, circuit breaker, and service mesh produced similar safe completion—77.43% [77.00%, 77.90%], 77.40% [76.97%, 77.87%], and 77.43% [77.00%, 77.90%]—because transport recovery alone could not substitute for missing evidence or authority. Their zero observed unsafe rates were achieved

through failed or blocked execution, whereas DMO-AI recovered low- and medium-risk work through constrained fallback.

6.3 Ablation results

Table 10
Ablations and paired DMO-AI advantage

Ablation	Safe completion	DMO gain	Unsafe	Violation	Blocked
No risk	91.50% [90.77%, 92.30%]	6.03 pp [5.27, 6.77]	3.50% [2.80%, 4.20%]	0.87% [0.50%, 1.23%]	3.10% [2.47%, 3.80%]
No local	82.67% [81.83%, 83.53%]	14.87 pp [14.30, 15.37]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	16.10% [15.33%, 16.80%]
No retrieval cache	93.87% [93.07%, 94.70%]	3.67 pp [3.03, 4.30]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	4.90% [4.33%, 5.50%]
No policy cache	89.90% [89.37%, 90.40%]	7.63 pp [6.97, 8.30]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	8.87% [8.23%, 9.57%]
No handoff	92.70% [92.03%, 93.43%]	4.83 pp [4.00, 5.60]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	6.07% [5.27%, 6.67%]

Removing task-risk semantics reduced safe completion to 91.50% [90.77%, 92.30%] and introduced 3.50% [2.80%, 4.20%] unsafe outcomes and 0.87% [0.50%, 1.23%] policy violations. This is the most direct evidence that the gain is not merely a larger fallback menu. Removing local inference caused the largest availability loss (82.67% [81.83%, 83.53%]), while removing policy cache, handoff, or retrieval cache reduced safe completion to 89.90% [89.37%, 90.40%], 92.70% [92.03%, 93.43%], and 93.87% [93.07%, 94.70%], respectively. The full policy combined those mechanisms without an observed safety penalty in the tested regime.

6.4 Descriptive results by risk and fault window

Aggregate safe completion can conceal whether a controller succeeds only on easy tasks. We therefore examined the frozen event logs by risk class. These analyses were not designated as additional primary comparisons and are reported descriptively. DMO-AI retained high safe completion in all three classes, whereas fail-open became less safe as task risk and dependency demand increased.

Table 11
Descriptive task-risk stratification in the primary confirmation

Risk	Tasks	DMO safe	Fail-open safe	DMO blocked	DMO handoff
Low	1663	99.10%	83.28%	0.12%	0.00%
Medium	910	94.29%	74.62%	2.75%	0.00%
High	427	98.36%	72.13%	1.64%	34.66%

Low-risk DMO-AI safe completion was 99.10%, compared with 83.28% for fail-open. For medium-risk tasks the corresponding values were 94.29% and 74.62%, and for high-risk tasks 98.36% and 72.13%. DMO-AI's medium-risk safe completion was lower than its low- and high-risk rates because those tasks frequently required several services yet were not always eligible for the high-risk handoff path. The high-risk result should not be read as evidence that high-risk deployment is generally safe; it reflects the reference policy's willingness to hand off rather than use local inference when critical dependencies were impaired.

Table 12
Descriptive DMO-AI outcomes by injected fault window

Fault window	Tasks	Safe completion	Blocked	Two most frequent modes
Healthy interval	1320	100.00%	0.00%	normal 99.5%, local model 0.2%
Model degraded	240	100.00%	0.00%	local model 86.2%, human handoff 13.8%
Retrieval outage	240	96.25%	3.33%	normal 52.9%, retrieval cache 35.4%
Policy outage	240	95.83%	4.17%	cached policy 42.5%, normal 40.4%
Tool degraded	240	99.17%	0.00%	normal 61.7%, read only 38.3%
All services degraded	240	94.58%	5.42%	read only 40.4%, cached policy 25.0%
Policy + tool outage	240	83.33%	1.25%	read only 32.5%, cached policy 28.8%
Model outage	240	100.00%	0.00%	local model 85.8%, human handoff 14.2%

DMO-AI safely completed every task in the model-degradation and model-outage windows by using local inference for eligible low- and medium-risk tasks and handoff for high-risk tasks. Retrieval and policy outages produced modest blocking when caches exceeded their risk-conditioned age limits. The simultaneous policy-and-tool outage was the most difficult window: safe completion fell to 83.33% because tasks that did not require both impaired services could proceed normally, while affected tasks were redirected to read-only, cached-policy, handoff, or blocked behavior. The result illustrates why the unit of control is the task's required dependency set rather than the global health vector alone.

6.5 Mode utilization, latency, and operational overhead

DMO-AI mean latency was 36.31 [34.54, 38.78] ms and p95 latency was 143.04 [133.14, 156.44] ms. Relative to fail-open, the paired mean-latency difference was − 9.59 [-10.87, -7.79] ms; relative to retry budget it was − 12.60 [-14.95, -9.96] ms. Mean mode-selection overhead was 13.64 [10.71, 18.19] µs. These measurements suggest that early fallback can offset selector overhead in this testbed, but they should not be generalized to multi-node throughput or production human-handoff latency.

Normal execution accounted for 58.1% of DMO-AI tasks. The most frequent substitute was local-model execution at 15.1%, followed by read-only at 9.0%, cached policy at 7.7%, human handoff at 4.9%, and retrieval cache at 3.9%. Only 1.1% were blocked. This distribution shows that the aggregate gain did not arise from a single fallback; it came from matching different service failures to different restricted behaviors.

Latency must be interpreted together with mode composition. Local computation, cache use, read-only execution, and early handoff can bypass a slow or unavailable network path, so a controller with additional policy logic can still have lower mean latency than a retry-oriented baseline. This does not imply that local models or human review are intrinsically fast. It shows only that the emulated fallback paths in this benchmark avoid repeated dependency waits.

6.6 Health sensing and telemetry extension

Table 13
Prespecified health-sensing extension (95% bootstrap confidence intervals)

Controller	Safe completion	Unsafe	Violation	Sensor FP	Sensor FN
DMO-AI	96.89% [96.57%, 97.20%]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	0.018% [0.000%, 0.044%]	0.039% [0.018%, 0.061%]
No health sensing	77.37% [77.10%, 77.63%]	0.00% [0.00%, 0.00%]	0.00% [0.00%, 0.00%]	Not applicable	Not applicable

Table 14
DMO-AI implementation overhead in the extension

Component	Mean [95% CI]	Unit
Mode selection	10.56 [9.63, 11.65]	µs/task
Audit emission	2.65 [2.51, 2.79]	ms/task
Audit traffic	226.11 [226.06, 226.15]	bytes/task
Policy + observed state	1989.60 [1989.38, 1989.85]	bytes/task

Across the ten extension pairs, DMO-AI safe completion was 96.89% [96.57%, 97.20%], compared with 77.37% [77.10%, 77.63%] when active sensing was removed. The paired DMO-AI advantage was 19.52 pp [18.96, 20.07]. DMO-AI's measured sensor false-positive and false-negative rates were 0.018% [0.000%, 0.044%] and 0.039% [0.018%, 0.061%], respectively. This supports active health sensing as a mechanism in the frozen fault schedule; it does not establish sensor performance under untested network noise, probe loss, or asynchronous multi-node failure.

Mean audit emission latency was 2.65 [2.51, 2.79] ms with 226.11 [226.06, 226.15] request/response bytes per task. The explicit policy matrix plus observed state occupied 1989.60 [1989.38, 1989.85] bytes in the Python process. These values show that the reference implementation's control metadata is small relative to model execution, while the audit HTTP call is a measurable cost that production systems should batch, sample, or emit asynchronously where permitted.

All 108,000 extension audit emissions succeeded. The near-zero sensor error rates reflect the deterministic injected states, the local network, and the chosen threshold; they should not be compared directly with production monitoring systems. The no-sensing ablation is intentionally severe: it receives an all-healthy vector and therefore continues normally unless another rule independent of observed failure applies. Its 19.52-point deficit demonstrates that the fallback menu alone does not produce resilience—the controller must receive sufficiently accurate and timely dependency observations.

6.7 Heterogeneous Kubernetes validation

Table 15
Outcomes in the genuine two-node K3s/Istio validation (20 paired seeds)

Controller	Safe completion	Unsafe	Commit	Mean ms	p95 ms
Direct	65.45% [64.20%, 66.75%]	5.45% [4.85%, 6.10%]	19.25% [17.55%, 20.90%]	137.99 [136.51, 139.70]	313.75 [311.44, 316.45]
Istio	64.70% [62.35%, 66.60%]	5.45% [4.85%, 6.10%]	18.90% [17.35%, 20.50%]	139.67 [134.64, 142.77]	309.52 [297.19, 316.95]
DMO-AI	93.90% [92.75%, 95.05%]	0.00% [0.00%, 0.00%]	21.45% [19.70%, 23.25%]	154.19 [152.17, 156.08]	315.24 [312.52, 318.16]
DMO-AI + Istio	93.95% [92.80%, 95.10%]	0.00% [0.00%, 0.00%]	21.35% [19.55%, 23.15%]	158.83 [154.60, 162.51]	311.22 [308.25, 314.13]

DMO-AI with Istio achieved 93.95% [92.80%, 95.10%] safe completion versus 64.70% [62.35%, 66.60%] for Istio, a paired gain of 29.25 pp [27.30, 31.85]. DMO-AI without mesh-specific retry similarly exceeded direct execution by 28.45 pp [27.15, 29.80]. Composing Istio with DMO-AI changed safe completion by only 0.05 pp [0.00, 0.15], whereas Istio changed the direct baseline by -0.75 pp [-2.25, 0.00]. Under this design, the dominant gain therefore came from semantic mode selection, not transport policy alone.

Mean end-to-end latency was 139.67 [134.64, 142.77] ms for Istio and 158.83 [154.60, 162.51] ms for DMO-AI with Istio. The approximately 19-ms mean increase is the observed cost of policy checks, transactional tool calls, and bounded edge fallback in this small sequential workload. It is not a throughput or saturation estimate. All successful real-model calls returned non-empty output, but the one-token setting means this observation verifies inference-path execution rather than answer quality.

Table 16
Descriptive outcomes by fault window in the multi-node validation

Window	Istio safe	DMO + Istio safe	Istio unsafe	DMO + Istio unsafe
healthy	93.8%	93.8%	0.0%	0.0%
model outage	0.0%	100.0%	0.0%	0.0%
model degraded	92.0%	97.3%	0.0%	0.0%
policy outage	63.7%	87.3%	36.3%	0.0%
tool outage	60.7%	97.0%	0.0%	0.0%
compound policy tool outage	58.7%	88.3%	0.0%	0.0%

DMO-AI with Istio used the edge model for 13.8% of all tasks, cached policy for 3.5%, read-only operation for 8.5%, and handoff for 4.0%. No high-risk task used edge substitution. Across all 80 controller–seed tool states, persisted reservation counts matched committed events; no negative inventory, duplicate replay, expired-cache commit, or integrity failure was observed. These are zero observed counts in this experiment, not proof that the corresponding population risks are zero.

7 Discussion

7.1 Why risk semantics matter

The baseline and ablation results support the central mechanism. A transport controller can know that a service is slow or unavailable, but it cannot infer whether cached evidence is fresh enough for the task, whether local inference is appropriate for its risk, or whether a write remains authorized. The no-risk ablation retained DMO-AI's mechanisms yet reintroduced unsafe outcomes. Risk semantics therefore function as a safety constraint on graceful degradation rather than an optional metadata feature.

The no-risk ablation is particularly informative because it retained local inference, caches, read-only behavior, handoff, and blocking. Its loss of safe completion and reappearance of unsafe and violating outcomes show that a larger menu of fallbacks is not sufficient. The controller must know which fallback is permitted for the consequence class of the current task. This finding separates DMO-AI from a generic graceful-degradation implementation that simply selects the fastest available substitute.

7.2 Availability need not require fail-open behavior

Fail-open and fail-closed are often treated as endpoints of an unavoidable trade-off. DMO-AI changes the decision space by adding constrained substitutes. Read-only operation preserves advice while suppressing a write; a fresh cache preserves grounding or policy evidence for a bounded period; and handoff preserves task progress without claiming autonomous authority. Safe completion can consequently increase even as unsafe continuation decreases.

The comparison also clarifies why raw completion is a poor objective for agent service chains. Fail-open completed more tasks than it completed safely, while fail-closed achieved zero observed unsafe outcomes by denying a large share of work. DMO-AI occupied a third region: it preserved much of the utility of continuation while changing the form of the task. A state-changing request can become advice; cloud inference can become a bounded local result; live evidence can become a fresh, explicitly uncertain cache; and autonomous control can become human review.

7.3 Deployment implications

Organizations should define degraded-mode policy before an incident: task-risk classes, cache-age limits, tool actions suppressed in read-only mode, local-model eligibility, retry budgets, and handoff targets. Audit events should link health state, task risk, selected mode, missing evidence, and outcome through the observability stack. The mode selector can run as agent middleware or a sidecar; service meshes should continue to enforce routing and transport policy rather than being asked to infer task semantics.

Implementation should begin with a dependency and authority inventory rather than with mode names. For each workflow, the operator must identify which services are necessary for correctness, which are necessary for authorization, which outputs can change external state, and which substitutes are acceptable at each risk level. Only then can cache windows, local-model eligibility, retry budgets, and handoff targets be defined. A fallback that is safe for document summarization may be unacceptable for financial execution or clinical communication.

Operational ownership must also be explicit. Platform teams can operate health probes, routing, and telemetry; application teams can define dependency semantics; risk or compliance owners can approve authority and freshness limits; and service owners can define recovery and handoff targets. The audit record should make these boundaries reviewable by including the policy version and reason code, not only the selected mode.

7.4 Sensing and observability are control inputs

The no-health-sensing extension distinguishes having fallback rules from knowing when they are applicable. Without active dependency observations, the controller cannot intentionally enter cache, local, read-only, or handoff modes in response to partial failure. Conversely, sensing and audit are not free: probes can misclassify transient states and synchronous audit emission adds latency. Production implementations should therefore define probe intervals and hysteresis, treat telemetry delivery as non-

authoritative for the task decision, and prevent observability failure from silently expanding agent permissions.

This creates a design tension. Aggressive probing improves state awareness but consumes network and service capacity, while slow probing increases stale-state risk. Synchronous auditing improves immediate trace completeness but adds critical-path latency and can couple task availability to the telemetry backend. Production designs should use risk-conditioned probe frequency, hysteresis around degradation thresholds, bounded stale-health windows, and asynchronous or buffered audit delivery where regulation and safety policy permit.

7.5 Evidence boundaries

Zero observed unsafe outcomes means zero events among 3,000 DMO-AI task executions in the primary confirmation and among 9,000 DMO-AI executions in the separate extension, plus zero events among 4,000 DMO-AI executions in the multi-node validation; it does not prove zero population risk. The controller is deterministic and the oracle operationalizes specified hazards. The result is evidence that the policy was internally consistent under the injected states, not proof of domain safety or regulatory compliance. A deployment must derive its risk taxonomy and authority rules from a domain-specific safety case.

The results also do not establish that the selected thresholds are optimal. Cache limits, probe thresholds, and the ordering of fallback rules are policy inputs. The experiment shows that one explicit policy behaved consistently under one frozen workload and that removing key mechanisms degraded the measured outcomes. It does not estimate the real-world probability or severity of every hazard, nor does it validate the semantic quality of local or cached outputs.

7.6 Relationship to cloud–edge orchestration

Most cloud–edge orchestration asks where and with what resources a task should execute. DMO-AI adds a logically prior question: what form of the task remains permissible under the current evidence and authority state? Placement and resource allocation can then optimize within that permitted form. A local edge node may have sufficient compute for an inference, for example, while policy still prohibits a lower-assurance model for a high-risk task. Conversely, a low-risk task may be allowed to run locally even when the globally optimal cloud route is unavailable.

This suggests a two-level controller. An offline or slower-timescale optimizer selects budgets, thresholds, replica placement, and capacity from expected workload and cost. A fast runtime assurance layer enforces feasibility and chooses among pre-approved modes for each task. The formal objective in Section 3 can support the first level, while Algorithm 1 implements the second. Future work can connect these levels by learning calibrated costs and failure probabilities without allowing learned estimates to override hard authority invariants.

The genuine deployment sharpens this separation. Istio and Envoy supplied named, real transport controls and observable sidecars, while DMO-AI supplied the task-semantic decision. Istio alone did not improve the deliberately availability-first application policy, and composing Istio with DMO-AI changed safe completion only marginally relative to DMO-AI alone. This is not evidence that service meshes lack value: the mesh enforced routing and produced telemetry, but it could not decide whether an absent policy service should convert a write into read-only behavior or handoff.

7.7 Practical adoption sequence

A cautious adoption sequence has four stages. First, run DMO-AI in shadow mode and compare its proposed decisions with existing incident and authorization policies. Second, enable non-state-changing modes such as rate limiting, cache use, and read-only output for low-risk tasks. Third, integrate bounded handoff and current policy evidence for higher-risk workflows. Fourth, extend the present two-node validation to randomized link faults, queue behavior, model quality, and human response before permitting broader autonomous substitution. At every stage, operators should retain a kill switch and versioned policy rollback.

8 Threats to Validity and Limitations

8.1 Construct validity

Construct validity is limited by synthetic tasks and an operational oracle. Required retrieval, policy, and tool evidence are generated from a probabilistic task model rather than extracted from a real application. Safety labels capture stale or missing evidence and authority, not semantic output quality, prompt injection, deception, or domain harm.

Safe completion is consequently a benchmark construct tied to enumerated hazards. It gives more information than raw completion, but it cannot represent every quality or safety dimension. The local-model and cached-retrieval modes are treated as safe when their policy conditions are satisfied; the experiment does not score factuality, calibration, or downstream user reliance. Human handoff is counted as completed after an acknowledgement and does not measure resolution.

8.2 Internal validity

Internal validity is strengthened by controller-blind labels, identical tasks and fault states, frozen code hashes, and paired analysis. Remaining threats include deterministic window ordering, sequential controller execution, health-sensing thresholds, and the chosen cache limits. The pilot-informed refinement was transparently separated from confirmation, but an independent replication is still needed.

Controllers execute sequentially rather than concurrently, reducing cross-controller interference but allowing slow host drift over the experiment. Common random numbers and paired analysis address workload variation, not every environmental confound. The health sensor uses the same dependency

endpoints for all controllers, but timing noise can create small differences between latent and observed state. These errors are measured in the extension rather than corrected with latent truth.

The multi-node infrastructure amendment is an additional internal-validity consideration. The first attempt lost its temporary volume when the edge node became unavailable; because no aggregate results were retained or inspected, the restart did not select a favorable observed outcome. Nevertheless, reducing each controller–seed block from 300 to 100 tasks and generation from eight to one token changed the workload after an operational failure. We therefore report the amendment, analyze only the complete restart, and avoid combining attempts.

8.3 External validity

External validity is limited to one single-host ARM64 container environment and one heterogeneous two-node topology. The Kubernetes validation used one AWS region, one local ARM64 edge VM, one lightweight model family, one Istio version, and one transactional tool. Faults were deterministic application-controlled conditions carried over real service paths; they were not randomized physical WAN impairment, autoscaler pressure, or multi-region partition. Accordingly, the study establishes mechanism behavior and measured service-chain overhead, not production scalability.

The retained benchmark does not deliberately reproduce Kubernetes pod eviction, autoscaling, node pressure, DNS failure, certificate expiry, message-queue backlog, multi-region inconsistency, or real wireless edge conditions. It also does not evaluate GPU model servers, vector databases, external policy engines, or transactional tools at production scale. The results should therefore be treated as mechanism evidence and a reproducible reference workload, not as a capacity plan.

8.4 Measurement validity

The extension's telemetry backend is a deterministic fifth HTTP service rather than a full OpenTelemetry collector, and the orchestration-state estimate is Python deep-object size rather than total resident or sidecar memory. Its sensor accuracy is conditional on the fixed 30-ms classification threshold and deterministic injected states. The first-stage service-mesh baseline is a transport-policy approximation implemented in the common harness; the separate Kubernetes validation addresses that boundary with Istio/Envoy 1.30.3 but uses a different, smaller workload.

Observed network bytes cover serialized request and response bodies and omit TCP/IP, TLS, proxy, and container-network headers. Process CPU and resident-memory measures are host-process observations rather than isolated per-mode energy or memory profiles. Decision overhead is measured around the selector function and excludes task classification, policy distribution, and external authorization latency. These choices make the reference implementation measurable but should not be extrapolated to total production cost.

8.5 Statistical conclusion validity

The two container stages use ten run-level pairs and the Kubernetes validation uses 20 seed-level pairs. Pairing yields precise contrasts for each frozen schedule, but neither design samples independent infrastructures and the intervals should not be interpreted as covering every workload regime. Future work should randomize fault-window order, vary prevalence and severity, run concurrent workloads across several cloud and edge sites, and test calibrated domain policies with larger local models and human operators.

The bootstrap intervals quantify variation across the ten seeds under a deterministic fault schedule. They do not sample hardware platforms, domains, policy matrices, or failure distributions. The stratified analyses use the same frozen data and were not multiplicity-adjusted because they are descriptive mechanism checks rather than new hypothesis tests. A stronger external study should preregister fault prevalence and severity, use more independent hosts, and report power and throughput under concurrent load.

9 Reproducibility

9.1 Artifact contents

The package contains Docker Compose service definitions, the Toxiproxy benchmark, controller implementations, two frozen protocols and their hashes, twenty task scenarios, 141,000 event rows across the two stages, per-run summaries, and the aggregation scripts. The primary and extension benchmark source hashes are a7dd32d987c2e3be225061e1b008da77e85b4202d2f1fd9026c7212d6cf355ca and 029d6dcf8db829bff51418c8db211107a04307d977525037cb7b97c9d1bffeb6. Pilot, validation, and confirmatory directories are separate, and the sensing extension is stored under its own root. A public archival link will be inserted before submission; until then, the package is available from the corresponding author for editorial inspection.

The multi-node artifact adds 8,000 event rows, 20 task seeds, 80 persisted tool-state audits, architecture-specific image checksums, K3s and Helm metadata, node and pod placement, deployment and Istio policy snapshots, Envoy statistics and cluster state, resource samples, WireGuard host evidence, the frozen protocol, Kubernetes manifests, and paired analysis outputs. Private keys and cloud credentials are excluded. The captured EC2 instance metadata is retained for provenance; the temporary instance, security group, EC2 key pair, and local SSH key were removed after collection.

For each frozen seed, the package includes the generated task scenario, event-level controller outcomes, run metadata, and controller summary. Aggregate directories contain run-level means, percentile bootstrap intervals, paired differences, and analysis metadata. Pilot and validation outputs remain available for audit but are segregated from confirmatory estimates. The release also includes citation metadata, licensing, a file manifest, and a standard SHA-256 checksum list.

9.2 Reproduction procedure

The primary confirmation can be rerun with Docker Compose using ten runs, 300 tasks per controller, and first seed 292001. The extension uses ten runs, 900 tasks per controller, and first seed 294001. Each primary event file must contain 3,300 controller–task rows, and each extension file 10,800 rows. The analysis scripts then recompute controller summaries and 5,000-resample paired intervals from the run directories.

A lightweight validation path does not require rerunning containers. The extension validator reads the packaged outputs, checks the expected seeds, controllers, tasks, audit status, aggregate run counts, comparisons, and frozen source hashes, and reports a structured valid or invalid result. Full-file integrity can then be checked against CHECKSUMS.sha256. These controls make accidental mixing of pilot, confirmatory, and extension evidence detectable.

The Kubernetes analysis can be reproduced offline by reading events.jsonl and tool_states.json from the multi-node result directory, recomputing the 80 seed–controller summaries, and applying the fixed 5,000-resample paired bootstrap with seed 94117. A full infrastructure reproduction uses the provided AWS provisioning, WireGuard/K3s join, multi-architecture image build, Istio installation, deployment, benchmark, evidence-collection, and teardown scripts.

10 Conclusion

Cloud–edge AI agents need a degraded-mode control plane that treats service availability and action authority jointly. In a frozen containerized fault-injection study, DMO-AI increased safe completion relative to five conventional resilience baselines while producing no observed unsafe outcomes or policy violations in the tested regime. Ablation showed that task-risk semantics were essential: removing them reintroduced both hazards. A separate frozen extension isolated active health sensing and quantified control-state and audit costs. The practical lesson is not that every agent should continue through failure, but that continuation, substitution, handoff, and blocking should be explicit, risk-bounded, observable runtime decisions. The additional heterogeneous Kubernetes study reproduced the direction and practical magnitude of the safe-completion advantage with a real service mesh, real lightweight inference, and a transactional state-changing tool.

The contribution is therefore both architectural and empirical. Architecturally, DMO-AI introduces the permitted degraded mode as an interface between cloud-native health controls and agent-specific safety policy. Empirically, the paired testbed shows that risk semantics, constrained substitutes, and active sensing jointly account for the observed improvement; the result is not reproduced by transport recovery, a permissive fallback menu, or policy rules without health observations.

The next validation step is replication across several cloud and edge sites with larger models and retrieval services, randomized and correlated network faults, concurrent load, power measurement, and actual human-handoff queues. Domain studies should then replace the synthetic oracle with calibrated safety cases. Until those steps are completed, DMO-AI should be understood as a reproducible runtime-control mechanism and testable design pattern, not a guarantee of production safety.

Declarations

Author Contribution

A.A.B. conceived and led the study; developed the DMO-AI framework, theoretical model, safety invariants, and degraded-mode orchestration architecture; designed the research methodology, workloads, fault schedules, controller configurations, ablation studies, and statistical analysis plan; implemented the containerised and heterogeneous Kubernetes test environments; configured the K3s, Istio/Envoy, WireGuard, cloud and edge inference services, and transactional SQLite tool; conducted the experiments; collected, validated, analysed, and interpreted the data; prepared the reproducibility materials, figures, tables, and initial manuscript draft; and coordinated all revisions. N.B. provided academic supervision, contributed to the refinement of the research design and theoretical framing, supported interpretation of the findings, and critically reviewed the manuscript. D.B. contributed technical and methodological feedback on the cloud–edge architecture, experimental interpretation, and manuscript review. All authors reviewed and approved the final manuscript and accept accountability for their respective contributions and the integrity of the published work.

Data Availability

The primary confirmation and prespecified extension workloads, raw event logs, per-run summaries, analysis outputs, container definitions, frozen protocols, source code, and complete multi-node Kubernetes evidence package will be deposited in a public repository. They are available from the corresponding author for editorial inspection.

References

1. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes. Commun. ACM. **59**(5), 50–57 (2016). https://doi.org/10.1145/2890784
2. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer (2017)
3. Sigelman, B.H., Barroso, L.A., Burrows, M., et al.: Dapper, a large-scale distributed systems tracing infrastructure. Google Res. (2010)
4. Oppenheimer, D., Ganapathi, A., Patterson, D.A.: Why do Internet services fail, and what can be done about it? In: USENIX Symposium on Internet Technologies and Systems (2003)
5. Dean, J., Barroso, L.A.: The tail at scale. Commun. ACM. **56**(2), 74–80 (2013). https://doi.org/10.1145/2408776.2408794
6. Beyer, B., Jones, C., Petoff, J., Murphy, N. (eds.): Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media (2016)

7. Nygard, M.T.: Release It! Design and Deploy Production-Ready Software. Pragmatic Bookshelf (2007)
8. Basiri, A., Blohowiak, A., Hochstein, L., Jones, N., Rosenthal, C.: Chaos engineering. IEEE Softw. **33**(3), 35–41 (2016). https://doi.org/10.1109/MS.2016.60
9. Rose, S., Borchert, O., Mitchell, S., Connelly, S.: Zero Trust Architecture. NIST SP 800 – 207 (2020). https://doi.org/10.6028/NIST.SP.800-207
10. Park, J.S., O'Brien, J.C., Cai, C.J., Morris, M.R., Liang, P., Bernstein, M.S.: Generative agents: interactive simulacra of human behavior. In: UIST (2023). https://doi.org/10.1145/3586183.3606763
11. Wang, L., Ma, C., Feng, X., et al.: A survey on large language model based autonomous agents. Front. Comput. Sci. **18**, 186345 (2024)
12. Xi, Z., Chen, W., Guo, X., et al.: The rise and potential of large language model based agents: a survey. Sci. China Inf. Sci. **68**, 121101 (2025)
13. Lewis, P., Perez, E., Piktus, A., et al.: Retrieval-augmented generation for knowledge-intensive NLP tasks. Adv. Neural Inf. Process. Syst. **33**, 9459–9474 (2020)
14. National Institute of Standards and Technology: Artificial Intelligence Risk Management Framework (AI RMF 1.0). NIST AI 100-1: (2023). https://doi.org/10.6028/NIST.AI.100-1
15. Foundation, O.W.A.S.P.: OWASP Top 10 for LLM Applications, version 2025 (2025). https://genai.owasp.org/llm-top-10/ Accessed 26 July 2026
16. Yakubov, D., Moreschini, S., Hästbacka, D.: Di-Select: a criteria-driven framework for selecting container orchestration distributions in cloud and edge computing. Cluster Comput. **29**, 478 (2026). https://doi.org/10.1007/s10586-026-06269-5
17. Shopify: Toxiproxy—A TCP proxy to simulate network and system conditions (2026). https://github.com/Shopify/toxiproxy Accessed 26 July 2026
18. Ullah, A., Dagdeviren, H., Ariyattu, R.C., Bowden, J., DesLauriers, J., Kiss, T.: MiCADO-Edge: towards an application-level orchestrator for the cloud-to-edge computing continuum. J. Grid Comput. **19**, 47 (2021). https://doi.org/10.1007/s10723-021-09589-5
19. Dähling, S., Razik, L., Monti, A.: Enabling scalable and fault-tolerant multi-agent systems by utilizing cloud-native computing. Auton. Agents Multi-Agent Syst. **35**, 10 (2021). https://doi.org/10.1007/s10458-020-09489-0
20. Elkhatib, Y., Poyato, P.: J.: An evaluation of service mesh frameworks for edge systems. In: 6th International Workshop on Edge Systems, Analytics and Networking, pp. 1–6. ACM (2023). https://doi.org/10.1145/3578354.3592867
21. Owotogbe, J., van den Kumara, I., Tamburri, D.A.: Chaos engineering: a multi-vocal literature review. ACM Comput. Surv. (2025). https://doi.org/10.1145/3777375
22. Ganguli, M., Ranganath, S., Ravisundar, S., Layek, A., Ilangovan, D., Verplanke, E.: Challenges and opportunities in performance benchmarking of service mesh for the edge. In: IEEE International

Conference on Edge Computing, pp. 78–85 (2021). https://doi.org/10.1109/EDGE53862.2021.00020

23. Yao, S., Zhao, J., Yu, D., et al.: ReAct: synergizing reasoning and acting in language models. In: International Conference on Learning Representations (2023). https://doi.org/10.48550/arXiv.2210.03629
24. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: language agents with verbal reinforcement learning. Adv. Neural Inf. Process. Syst. **36** (2023). https://doi.org/10.48550/arXiv.2303.11366
25. Schick, T., Dwivedi-Yu, J., Dessì, R., et al.: Toolformer: language models can teach themselves to use tools. Adv. Neural Inf. Process. Syst. **36** (2023). https://doi.org/10.48550/arXiv.2302.04761
26. Gupta, A.: ReliabilityBench: evaluating LLM agent reliability under production-like stress conditions. arXiv:2601.06112 (2026). https://doi.org/10.48550/arXiv.2601.06112
27. Zhan, Q., Liang, Z., Ying, Z., Kang, D.: InjecAgent: benchmarking indirect prompt injections in tool-integrated large language model agents. In: Findings of ACL 2024, pp. 10471–10506 (2024). https://doi.org/10.18653/v1/2024.findings-acl.624
28. Debenedetti, E., Zhang, J., Balunovic, M., Beurer-Kellner, L., Fischer, M., Tramèr, F.: AgentDojo: a dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. Adv. Neural Inf. Process. Syst. **37** (2024). https://doi.org/10.52202/079017-2636
29. Ruan, Y., Dong, H., Wang, A., et al.: Identifying the risks of LM agents with an LM-emulated sandbox. In: International Conference on Learning Representations (2024). https://doi.org/10.48550/arXiv.2309.15817
30. Sha, L.: Using simplicity to control complexity. IEEE Softw. **18**(4), 20–28 (2001). https://doi.org/10.1109/MS.2001.936213
31. Mehmood, U., Roy, S., Damare, A., Grosu, R., Smolka, S.A., Stoller, S.D.: A distributed Simplex architecture for multi-agent systems. J. Syst. Archit. **134**, 102784 (2023). https://doi.org/10.1016/j.sysarc.2022.102784
32. Chen, S., Sun, Y., Li, D., Wang, Q., Hao, Q., Sifakis, J.: Runtime safety assurance for learning-enabled control of autonomous driving vehicles. arXiv:2109.13446 (2021). https://doi.org/10.48550/arXiv.2109.13446
33. Patel, Y.S., Townend, P., Singh, A., Östberg, P.-O.: Modeling the Green Cloud Continuum: integrating energy considerations into cloud–edge models. Cluster Comput. **27**, 4095–4125 (2024). https://doi.org/10.1007/s10586-024-04383-w
34. Alva, L.R., Pandey, B.: Agentic AI systems in the age of generative models: architectures, cloud scalability, and real-world applications. Artif. Intell. Rev. **59**, 88 (2026). https://doi.org/10.1007/s10462-025-11458-6
35. Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., Mané, D.: Concrete problems in AI safety. arXiv:1606.06565 (2016). https://doi.org/10.48550/arXiv.1606.06565
36. Rancher Labs: K3s—Lightweight Kubernetes. https://docs.k3s.io/ Accessed 27 July 2026

37. Donenfeld, J.A.: WireGuard: next generation kernel network tunnel. In: Network and Distributed System Security Symposium (2017). https://www.wireguard.com/papers/wireguard.pdf
38. Istio, Authors: Istio 1.30 documentation. https://istio.io/latest/docs/ Accessed 27 July 2026
39. Gerganov, G., et al.: llama.cpp—LLM inference in C/C++. https://github.com/ggml-org/llama.cpp Accessed 27 July 2026
40. SQLite Consortium: SQLite documentation. https://www.sqlite.org/docs.html Accessed 27 July 2026

Figures

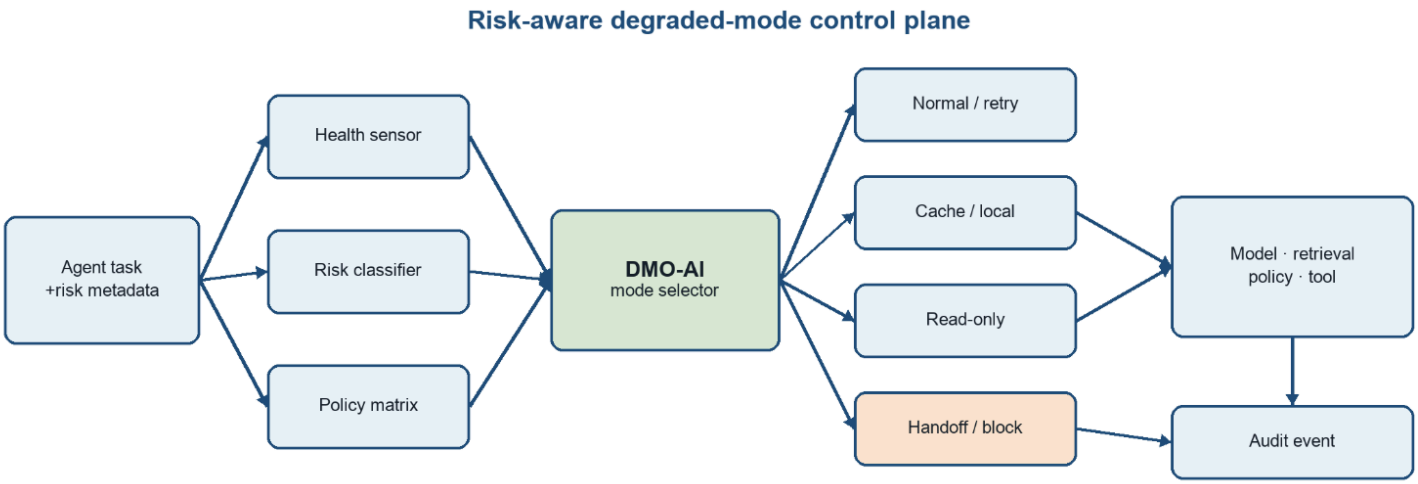


Figure 1

DMO-AI risk-aware degraded-mode control plane

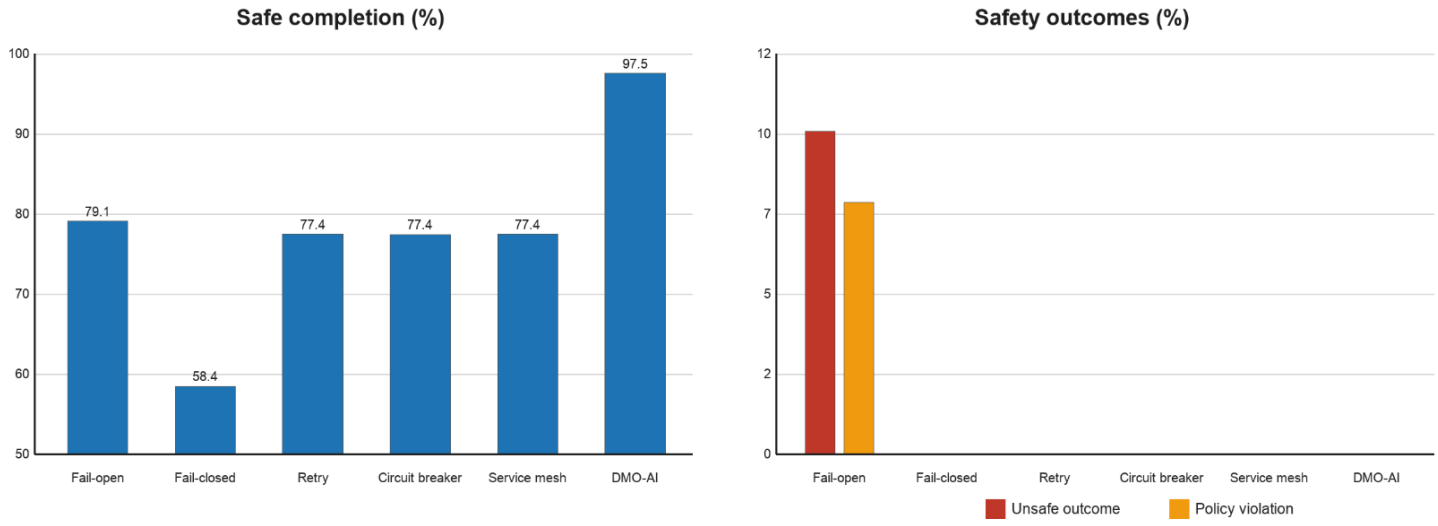


Figure 2

Safe completion and safety outcomes in the confirmatory testbed

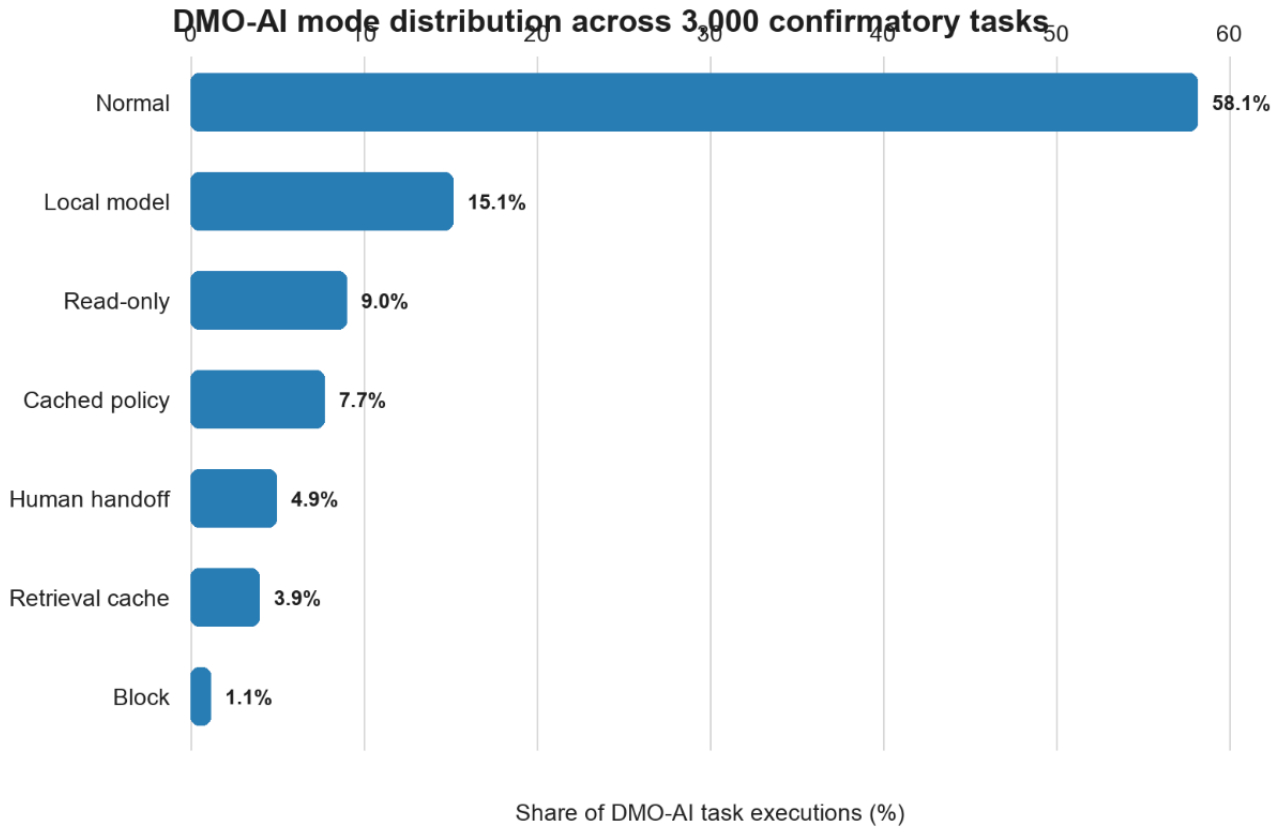


Figure 3

Distribution of DMO-AI modes in the primary confirmation

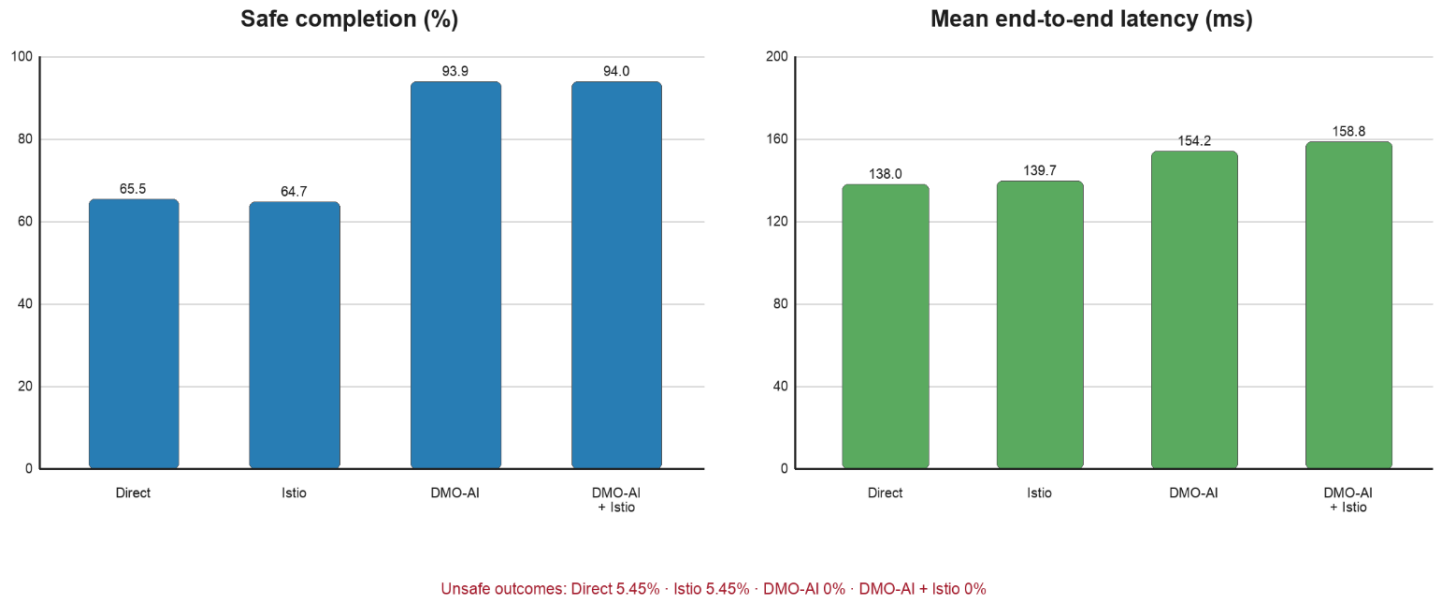


Figure 4

Safe completion, latency, and observed safety outcomes in the multi-node validation