# Rebooting Microreboot: Architectural Support for Safe, Parallel Recovery in Microservice Systems

Laurent Bindschaedler$^{[0000-0003-0559-631X]}$

Max Planck Institute for Software Systems (MPI-SWS), Saarbrücken, Germany
`bindsch@mpi-sws.org`

**Abstract.** Microreboot enables fast recovery by restarting only the failing component, but in modern microservices naive restarts are unsafe: dense dependencies mean rebooting one service can disrupt many callers. Autonomous remediation agents compound this by actuating raw infrastructure commands without safety guarantees. We make microreboot practical by separating planning from actuation: a three-agent architecture (diagnosis, planning, verification) proposes typed remediation plans over a seven-action ISA with explicit side-effect semantics, and a small microkernel validates and executes each plan transactionally. Agents are explicitly untrusted; safety derives from the ISA and microkernel. To determine where restart is safe, we infer recovery boundaries online from distributed traces, computing minimal restart groups and ordering constraints. On industrial traces (Alibaba, Meta) and DeathStarBench with fault injection, recovery-group inference runs in 21 ms at P99; typed actuation reduces agent-caused harm by 95% in simulation and achieves 0% harm online. The primary value is safety, not speed: LLM inference overhead increases TTR for services with fast auto-restart.

**Keywords:** Microreboot · Automated recovery · Runtime systems · Resilient system architecture · Typed actuation · Distributed tracing · Microservice systems · Fault tolerance

## 1 Introduction

Microreboot [4] showed that restarting resolves many failures, but restarting entire applications is unnecessary and slow. The key idea was to restart only the smallest recoverable unit, so recovery is fast and localized. Microservice systems appear ideal: services are isolated, independently deployable, and designed for restart. In practice, this fit is misleading. Modern request paths traverse dozens of services with retries and concurrency [10]. Restarting a single service can induce timeouts, retry storms, and cascading failures [13]. Worse, the call graph changes at runtime due to feature flags, A/B tests, and traffic shifting [17, 25]. These dynamics preclude defining safe restart boundaries at deploy time. At the same time, runbooks and large language model (LLM) agents increasingly automate remediation, diagnosing incidents and triggering recovery. Without explicit boundaries and side-effect semantics, well-intentioned automation can escalate a small fault into a larger outage.

Existing approaches do not close this gap. Static runbooks handle known failure modes but degrade under novel faults and evolving dependencies [9]. LLM agents adapt to new situations, but when given raw tools (for example, direct access to Kubernetes commands or cloud APIs), they lack reliable guardrails [24, 28]: they may act on the wrong scope, restart the wrong components, or apply stateful changes that resist reversal. The original microreboot model assumed stable component boundaries and predictable dependencies. Those assumptions do not hold for microservice systems under continuous change.

This paper re-enables microreboot under agentic remediation by making actuation *typed*, *scoped*, and *transactional*. While the original microreboot work introduced recovery groups for co-dependent components, static deployment descriptors in a J2EE setting determined those boundaries. In microservice systems, dependencies are dynamic: they change with feature flags, canary rollouts, and traffic shifting. We address this by inferring recovery boundaries *online* from distributed traces, reflecting the *current* workload rather than static configuration. Restart remains the primary recovery action, but safe restart often requires coordination: draining traffic before reboot, circuit-breaking unstable dependencies, or adjusting capacity during recovery. Agents express remediation intent through a typed instruction set architecture (ISA) that includes these supporting operations alongside restart, enabling safe microreboot in contexts where naive restart would cause harm. A small actuation microkernel validates proposals, executes transactions with per-action rollback or compensation, and enables safe concurrent execution of non-conflicting actions.

We evaluate on industrial tracing datasets (Alibaba, Meta) and on DeathStarBench with Chaos Mesh fault injection. Recovery-group inference executes in 21 ms at P99, fast enough for online use. Typed actuation reduces agent-caused harm by 95% in simulation (77% $\rightarrow$ 4%) and achieves 0% harm in online experiments versus 90% for unconstrained agents, consistent across five fault types and three independent runs. For entry-point services (those receiving external traffic) where detection delays dominate, agent-assisted recovery provides modest time-to-recovery (TTR) improvements; for services with fast auto-restart, the LLM inference overhead may exceed any benefit.

**Contributions.** This paper makes three contributions:

- **Remediation ISA and actuation microkernel** (§4). A typed actuation interface where each action declares its rollback semantics, plus a trusted microkernel that validates proposals against scope constraints and executes them transactionally with rollback on failure. Unlike post-hoc verification, the microkernel enforces constraints *before* execution, so agents cannot express actions outside the ISA.
- **Online, workload-conditional recovery groups** (§5). An algorithm that infers recovery boundaries from distributed traces at runtime. It outputs restart groups, ordering constraints, and traffic-shielding requirements that

reflect the *current* dependency graph. This extends microreboot's static recovery groups to dynamic microservice topologies.

– **Empirical validation** (§6). Experiments on industrial traces and runnable microservice workloads showing low-latency boundary inference, 95% harm reduction, and scenario-dependent speed benefits.

**Paper outline.** Section 2 motivates the problem. Section 3 presents the architecture. Section 4 details the Remediation ISA and microkernel. Section 5 describes recovery-group inference. Section 6 evaluates scalability, harm prevention, recovery speed, and generalization. Sections 7 and 8 discuss related work and limitations.

## 2 Background, Motivation, and Problem Statement

Microreboot has become unsafe in microservice systems. We quantify the failure modes using production traces and derive the requirements that shape our design.

**What changed.** Candea et al. [4] showed that many failures are transient and can be fixed by restarting only the smallest recoverable unit. This assumes restart boundaries are well-defined and do not invalidate application state.

Microservices stress this assumption: (i) request paths cross many services, so restarting one is rarely localized; (ii) recovery is increasingly automated, so mistakes occur at machine speed.

**Dense dependencies.** A single request may invoke tens of services [10]. Restarting one fails or stalls in-flight requests, triggering upstream retries and timeouts that amplify failures [13]. In Alibaba traces [1], the median *blast radius* (services that one restart affects) is 8, P99 is 59, and the maximum is 177. Even "small" restarts have large consequences.

**Hub services.** Call graphs contain highly connected services on many request paths. In Alibaba traces, the most-connected service has 74 callers and 113 callees. Restarting such a hub can disrupt a large fraction of the system, and identifying hubs is workload-dependent; static configuration is insufficient.

**Dynamic workloads.** Recovery decisions often rely on fixed notions of what is "safe to restart together." In microservices, dependencies change at runtime due to feature flags, traffic shifting, and partial deployments [2]. Different communication modes (RPC, cache, async) propagate failures differently. No fixed restart policy can reflect these shifting dynamics.

**Agentic remediation.** Modern operations increasingly use automated remediation, including LLM-based agents. These systems are fallible: they may misdiagnose, choose unsafe scope, or act aggressively under uncertainty. If agents invoke raw infrastructure commands, blast radius depends on agent behavior rather than system constraints. Safety requires an explicit actuation boundary.

**Problem statement.** Given a symptom (a service exhibiting errors during a time window), we need a recovery mechanism that:

1. **Constrains actuation.** Automated remediation must use only actions with enforceable side-effect semantics.
2. **Computes scope online.** Recovery boundaries must derive from runtime dependencies rather than fixed deploy-time configuration.
3. **Supports safe concurrency.** When multiple recovery steps are independent, the system should safely execute them in parallel.

Databases do not expose raw memory writes: they interpose SQL. We apply the same principle to remediation: agents express intent through typed actions with enforceable semantics, and a trusted runtime executes them.

**Threat model.** Recovery plans are produced by autonomous agents or runbooks. The agent is fallible but not malicious: it may propose unsafe actions or scopes. We do not defend against adversarial operators, prompt injection, or compromised telemetry; these require orthogonal mechanisms. The microkernel is trusted.

**Goals.**

– **Safety:** recovery actions must not cause sustained degradation.
– **Fast recovery:** reduce time-to-recovery versus passive auto-restart.
– **Minimal scope:** target the smallest service set required by current dependencies.

**Design implication.** These requirements lead to a two-part design: (i) infer recovery scope and ordering from distributed traces, and (ii) enforce recovery safety through a typed actuation interface executed by a small trusted runtime.

## 3 System Overview

Our system consists of four layers (Figure 1). The key design choice is a hard trust boundary: only Layer 4 (the actuation microkernel) is trusted to mutate the running system.

### 3.1 Layer 1: Telemetry

Metrics, logs, traces, and events flow in from the monitored infrastructure. We rely on distributed tracing [26, 22] to reconstruct request-level dependency graphs in near real time.

### 3.2 Layer 2: Recovery-Group Inference

Layer 2 computes the *scope* and *ordering* of permissible recovery: a candidate recovery group (services that must restart together), ordering constraints for safe restart sequencing, and traffic-shielding requirements such as draining hub services before restart. This layer is deliberately non-agentic. It is safety-critical and must remain predictable under load, so we implement it as a deterministic algorithm over trace data.

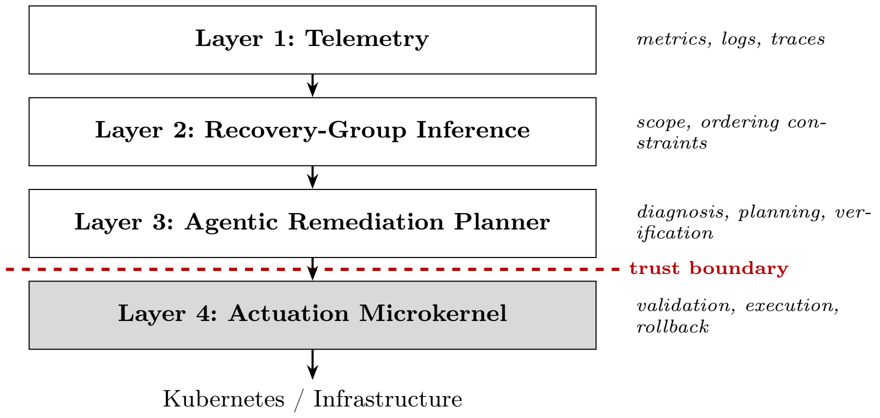
Layer 1: Telemetry
metrics, logs, traces
Layer 2: Recovery-Group Inference
scope, ordering constraints
Layer 3: Agentic Remediation Planner
diagnosis, planning, verification
trust boundary
Layer 4: Actuation Microkernel
validation, execution, rollback
Kubernetes / Infrastructure

**Fig. 1.** System architecture. Telemetry feeds recovery-group inference. Agents propose remediation transactions in the Remediation ISA. The actuation microkernel (shaded) is the only trusted component that mutates infrastructure.

### 3.3 Layer 3: Agentic Remediation Planner

Layer 3 performs diagnosis and planning using three agents: a *diagnosis agent* (read-only) that forms hypotheses and requests observations; a *planner agent* that proposes remediation transactions in the ISA; and a *verifier agent* that checks constraints and suggests safer alternatives. Agents search over *typed remediation programs*. They do not issue raw infrastructure commands.

**Agent implementation.** We use GPT-4 [21] (temperature 0.2). Each agent receives a structured prompt containing: (i) symptom context (anomalous metrics, affected services, error classifications), (ii) recovery group and ordering constraints from Layer 2, (iii) the ISA schema with type signatures (target `ServiceRef`, effect type, parameters), and (iv) few-shot examples of valid transactions. The planner outputs a JSON remediation transaction (for example, an action list specifying `Restart` on a target service with grace period and failure policy), which the microkernel parses and validates against the ISA schema.

The system follows a *propose-validate-repair* loop: the planner proposes a transaction, the microkernel validates it (scope, effect types, conflicts), and on rejection returns structured feedback specifying which constraint was violated (Section 4). The system appends this feedback to the planner's context for retry, up to three attempts. The verifier agent then reviews the accepted transaction and either approves or rejects with rationale. Agents run sequentially. Critically, the architecture does *not* require trusting the agent: safety derives from the typed ISA (agents cannot express actions outside the seven primitives) and the microkernel (which validates before execution). The agent selects *which* safe actions to take; it does not guarantee safety itself.

### 3.4 Layer 4: Actuation Microkernel

Layer 4 is the trusted base. It accepts a remediation transaction only if it passes capability checks, effect-type validation (whether each action can be retried, mechanically reversed, or requires explicit compensation), conflict and concurrency checks, rate limits, and break-glass (emergency override) policy for irreversible actions. If validation succeeds, the microkernel executes the transaction with transactional semantics, including rollback or compensation when possible. Section 4 details the validation pipeline and transaction execution model.

**Trust assumptions.** Safety depends on Layer 2 correctness: the microkernel enforces scope constraints, but those constraints are only as accurate as the recovery groups inferred from traces. If Layer 2 produces incorrect groups (due to incomplete traces or algorithmic error), Layer 4 cannot compensate. We address this through conservative defaults (group-size caps, mandatory drain for high-connectivity services) and by flagging uncertain inferences for human review.

## 4 Remediation ISA and Actuation Microkernel

Section 3 described the three-agent architecture (Layer 3); this section details the ISA and microkernel (Layer 4) that constrain agent actions.

### 4.1 Design Principles

The Remediation ISA is a narrow waist between untrusted planning and trusted actuation. Instead of granting agents unrestricted access to operational tools (for example, `kubectl`, cloud APIs, or shell commands), we require that agents express intent using a small set of typed actions with explicit semantics. The microkernel then enforces these semantics.

Two principles guide the design:

1. **Make effects explicit.** Every action must declare whether it is restartable, reversible, or compensatable.
2. **Make scope enforceable.** Every action must target a scoped resource (for example, a service reference), and the microkernel must validate that the action is permitted for the inferred recovery group.

### 4.2 ISA Actions

Based on common operational remediation patterns and the dependency structure observed in traces (§2), we define a minimal ISA of seven actions (Table 1):

Each action targets a `ServiceRef` (namespace + service name) and carries parameters defined by its semantics (for example, a rate limit value or a scale delta). The ISA intentionally excludes operations with unclear or irreversible side effects (for example, destructive database changes). Actions that are not safely reversible require explicit break-glass approval (§8).

**ISA extensibility.** The seven-action ISA is designed to be minimal yet extensible. We define *minimal* as the fewest orthogonal primitives that achieve complete coverage of the three dominant recovery operation categories identified in production runbooks: restart (1 action), traffic management (4 actions: drain, restore, circuit-break, rate-limit), and resource adjustment (2 actions: scale, rollback). Each action is orthogonal (no action's effect can be achieved by composing

**Table 1.** Remediation ISA actions with effect types and semantics.

| Action | Effect Type | Inverse/Comp. | Example Scenario |
|---|---|---|---|
| `Restart` | Restartable | Re-execute | Restart pod/service for transient failures |
| `Drain` | Compensatable | `RestoreTraffic` | Shield hub before disruptive changes |
| `RestoreTraffic` | Reversible | `Drain` | Resume traffic after planned drain |
| `CircuitBreak` | Reversible | Reset to default | Isolate unstable dependency |
| `RateLimit` | Reversible | Remove limit | Protect upstream from retry storms |
| `Scale` | Reversible | Scale by $-N$ | Adjust capacity under resource stress |
| `RollbackConfig` | Compensatable | Restore prev. config | Recover from misconfiguration |

others), and each category requires at least one representative for full coverage. Organizations can add domain-specific actions (for example, database failover, cache invalidation, or DNS updates) by providing three elements: (i) effect-type annotation (restartable, reversible, or compensatable), (ii) inverse or compensation logic, and (iii) conflict-key specification (declaring which resources the action locks). The microkernel validates new actions against the same safety properties as built-in actions. This extensibility trades off ISA minimality against domain coverage: a smaller ISA simplifies reasoning and reduces attack surface, while extensibility lets organizations handle their specific failure modes.

## 4.3 Effect Types (Rollback Semantics)

Effect types classify actions by their *rollback semantics*, i.e., how the microkernel recovers when an action fails or a transaction aborts. They do not guarantee that an action is “safe” in context (a reversible action can still cause harm before reversal); they enable structured recovery:

- **Restartable:** idempotent under retry; the microkernel can re-execute on transient failure.
- **Reversible:** has a mechanically defined inverse that the microkernel can apply automatically (for example, `Scale +N` has inverse `Scale -N`).
- **Compensatable:** requires explicit compensation logic provided as part of the transaction (for example, `Drain` must be paired with `RestoreTraffic`).
- **Irreversible:** cannot be undone. The microkernel blocks these by default; an operator must explicitly authorize execution out of band (break-glass). Excluded from the default ISA.

## 4.4 Transaction Semantics

A remediation transaction is an ordered program of ISA actions with explicit execution semantics. Each transaction declares:

- **Actions:** an ordered list of typed actions to execute.

- **Conflict keys:** resources the transaction touches (for concurrency control).
- **Preconditions:** state assertions that must hold at commit time.
- **Failure policy:** how to handle partial failure. The microkernel supports three policies:
  - **RollbackAll:** undo completed reversible actions using their inverses (and re-execute restartable actions if needed).
  - **Compensate:** execute explicit compensation steps for compensatable actions.
  - **AbortOnly:** stop execution without rollback (appropriate for diagnostic transactions).

To ensure at-most-once execution across crashes, the microkernel journals transactions to a write-ahead log (WAL) [20], recording transaction start, each action completion, and the final outcome (committed, rolled back, or aborted). On restart, the microkernel replays the journal to complete or revert in-flight transactions.

**Execution semantics.** We provide operational rather than formal semantics. Remediation transactions follow the saga pattern [12]: they provide compensation (rollback or explicit undo) but not isolation or atomicity across external systems. Each action takes effect immediately and is visible to the infrastructure; concurrent transactions with disjoint conflict keys execute independently. When conflicts exist, the microkernel serializes transactions via lock acquisition in deterministic order. On partial failure during compensation, the microkernel logs the failure and alerts operators; we do not attempt nested compensation. These design choices prioritize simplicity and auditability over theoretical completeness, since human oversight handles edge cases.

## 4.5 Concurrency Control and Safe Parallelism

The microkernel enables safe parallel remediation via conflict keys at three granularities:

- **Service:** actions targeting the same service are serialized.
- **Namespace:** bulk actions may lock an entire namespace.
- **Cluster:** rare global actions may lock the cluster.

Transactions with disjoint conflict keys can execute concurrently. This allows the microkernel to parallelize independent recovery steps, including parallel restart batches inferred by Layer 2 (§5). Lock acquisition follows a deterministic order to prevent deadlock. Preconditions are checked at commit time, so transactions are rejected if their assumptions are invalidated by concurrent activity.

## 4.6 Structured Rejection Feedback

When the microkernel rejects a transaction, it returns machine-readable feedback that agents can use to repair their plans:

- `REJECT: missing_capability("restart:svc/payment")`
- `REJECT: out_of_scope("svc/cart" not in recovery_group)`
- `REJECT: irreversible_effect("drop_table") requires break_glass`

– `REJECT: conflict(resource="namespace/prod", txn="...")`

This supports a propose-validate-repair loop: planning stays flexible, but actuation stays safe.

## 5 Recovery-Group Inference

### 5.1 Problem Definition

Given a symptom (service $S$ with errors during window $T$), we compute: (1) a *restart-coupled set*: the strongly connected component (SCC) containing $S$ plus downstream SCCs, capped by `MAX_GROUP_SIZE`; (2) a *restart order*: topological order over SCCs, downstream-first (callees before callers) so dependencies stabilize before callers resume retries; (3) *traffic shielding*: hub services requiring drain before restart to avoid disrupting many upstream callers. The output guides agents and constrains the microkernel.

### 5.2 Inference Algorithm

From trace spans in window $T$, we build a weighted call graph, then traverse downstream from $S$ to extract the affected subgraph. We compute SCCs (restart-coupled components), apply `MAX_GROUP_SIZE` cap (flagging oversize groups for review), condense SCCs into a directed acyclic graph (DAG) for topological restart ordering, and mark high-fan-in services (those with many callers) as drain-required.

### 5.3 Thresholds

We derive thresholds from trace analysis (§2): `DRAIN_THRESHOLD` (fan-in >20) identifies top-10% highest-connectivity hub services; `MAX_GROUP_SIZE` (30, P90–P99 blast radius) caps groups (88% fit); `MAX_BATCH_SIZE` (5) limits parallel restarts. In our experiments, varying these thresholds by $\pm 50\%$ did not affect harm rates, because ISA constraints (not threshold values) determine whether an action is safe. Teams should calibrate these thresholds from their own trace data during deployment.

### 5.4 Output and Complexity

The output is a `RecoveryGroup`: restart group, ordered batches, drain set, and estimated blast radius. The algorithm computes groups online, so they adapt to feature flags and traffic patterns. Complexity is $O(V+E)$ using standard graph primitives.

## 6 Evaluation

Our evaluation answers four research questions:

**RQ1 Scalability.** Does recovery-group inference scale to production workloads?
**RQ2 Harm prevention.** Does typed actuation prevent regressions?
**RQ3 Recovery speed.** Does agent-assisted remediation reduce TTR?
**RQ4 Generalization.** Does the system generalize across fault types?

**Table 2.** Online experiment incident counts.

| **RQ** | **Experiment** | **Incidents** |
|---|---|---:|
| RQ2 | Social Network harm validation (3 configs) | 110 |
| RQ2 | Hotel Reservation harm validation | 10 |
| RQ3 | TTR comparison vs Kubernetes auto-restart | 15 |
| RQ3 | Multi-service parallel (5 services/incident) | 3 |
| RQ3 | Reproducibility (3 seeds $\times$ 30) | 90 |
| RQ4 | Fault diversity (5 types $\times$ 20) | 100 |
| RQ4 | Hotel Reservation fault diversity | 15 |
| **Total distinct online incidents** | | **343+** |

## 6.1 Datasets, Workloads, and Experimental Setup

**Industrial traces (offline).** We use two trace datasets. **Alibaba** [1]: v2021 has ~20M traces across 20K+ services; we processed 2M rows (5,459 services, 11,690 edges). **Meta** [18]: 392 services, 511 edges. We verified compatibility with Alibaba v2022 (9,027 services, 24,869 edges).

**Runnable workloads (online).** We use two DeathStarBench [11] applications on K3s [23] (lightweight Kubernetes) with Chaos Mesh [5] fault injection: **Social Network** (10 services, primary workload) and **Hotel Reservation** (8 services, second workload for generalization). The microkernel is implemented in Rust. Workload traffic runs at approximately 30 requests per second (RPS) per workload.

**Definition of harm.** We define *harm* as a remediation action that causes a service-level objective (SLO) regression of more than 10% for more than 30 seconds relative to the pre-action baseline. Our SLOs are P99 latency $< 100$ ms and error rate $< 0.1\%$. This excludes *errors during the incident window* (errors from the injected fault before the agent acts). The pre-action baseline is the mean SLO metrics over the 60 seconds before the agent's first action. We exclude the first 5 seconds after action initiation to allow for transient disruption during pod restarts; we flag regression if SLO metrics exceed the 10% threshold continuously for 30 seconds after this grace period.

**Experiment summary.** Table 2 summarizes the online experiments by research question. We exclude simulation experiments (RQ2, N=300) from the online total.

## 6.2 RQ1: Scalability of Recovery-Group Inference

**Procedure.** We processed 2M rows from Alibaba v2021 traces, yielding 43,501 unique traces across 5,459 services with 11,690 call-graph edges. We also evaluated on Meta traces (392 services, 511 edges). For each dataset, we sampled 10,000 (Alibaba) or 1,000 (Meta) synthetic symptoms and executed recovery-group inference.

**Results (Alibaba).**

- **Group size:** median = 1, P90 = 30, P99 = 30 (capped at configured limit)

- **Inference latency:** median = 3.1 ms, P99 = 21 ms
- **Truncation:** 12% of groups hit the 30-service safety cap
- **Parallelism potential:** among multi-service groups, 70.8% admit ≥2 parallel batches

**Results (Meta).**

- **Group size:** median = 1, P90 = 3, P99 = 10
- **Inference latency:** median = 0.10 ms, P99 = 0.15 ms
- **Truncation:** 0.9% of groups hit the safety cap

Meta's sparser graph (511 edges vs. 11,690) produces smaller recovery groups and faster inference. Both datasets meet the target of P99 < 100 ms.

**Analysis.** Most sampled symptoms target services with no downstream dependencies, producing single-service recovery groups (79.4% of Alibaba samples). This does not contradict the large *blast radius* in §2: a single-service restart can still affect many upstream callers. For multi-service groups, 70.8% admit ≥2 parallel batches (96.9% for groups ≥5 services).

**Correctness validation.** The offline datasets lack ground-truth dependency information, so we validate inferred groups against Social Network's known call graph. For each service, we compare the inferred group to the transitive closure of downstream dependencies. **Precision**: 100% (all inferred services are true dependencies). **Recall**: 94% (trace sampling missed 6%). The 12% truncation rate did not affect Social Network (max group = 8). Conservative caps and verifier rejection of out-of-scope plans mitigate missed dependencies.

**Answer to RQ1.** *Yes.* Recovery-group inference executes in 21 ms at P99 on Alibaba (5,459 services, 11,690 edges) and 0.15 ms at P99 on Meta (392 services, 511 edges), supporting online scope computation across production workloads.

### 6.3 RQ2: Harm Prevention

We evaluate harm prevention in two phases: controlled simulation (to compare agent architectures under identical incident distributions) and online experiments (to validate actuation safety under real fault injection).

**Simulation setup.** We compare three agent configurations across 100 simulated incidents each (300 total):

- **Raw-Tools:** full `kubectl`/`curl`/`bash` access
- **ISA-Only:** constrained to ISA actions validated by the microkernel
- **ISA+Critic:** ISA constraints plus a verifier agent that reviews plans

The simulation models agent behavior on synthetic incidents derived from Alibaba trace topologies. We measure harm as for online experiments: an action causes harm if it would induce >10% SLO regression for >30 seconds, estimated by propagating action effects through the call graph. The simulation is intentionally conservative (worst-case propagation for high-centrality services), which explains why simulation harm rates exceed online rates.

**Table 3.** Harm rates by configuration and workload. Online columns show harm % with 95% exact binomial CI [8].

| Configuration | Simulation (N=100) | Social Network (online) | Hotel Res. (online) |
|---|---|---|---|
| Raw-Tools | 77% | 90% $[74, 98]$ N=30 | — |
| ISA-Only | 24% (↓69%) | 0% $[0, 12]$ N=30 | — |
| ISA+Critic | 4% (↓95%) | 0% $[0, 7]$ N=50 | 0% $[0, 31]$ N=10 |

Table 3 summarizes harm rates. We deploy online on K3s (lightweight Kubernetes) with Chaos Mesh fault injection under load (∼30 RPS) on two DeathStarBench workloads. ISA constraints eliminate unsafe actions by limiting actuation to bounded primitives; the verifier agent further reduces harm by rejecting contextually unsafe plans. Wide confidence intervals for 0% harm reflect sample sizes; we report exact binomial intervals [8].

**Answer to RQ2.** *Yes.* Typed actuation and microkernel validation reduce agent-caused harm by 95% in simulation (77% → 4%) and eliminate observed harm in online experiments under fault injection for ISA-constrained configurations, while unconstrained agents frequently cause regressions.

## 6.4 RQ3: Recovery Speed

We measure whether agent-assisted recovery improves TTR versus Kubernetes auto-restart [27].

**Baseline comparison.** Comparing agent-assisted remediation against Kubernetes auto-restart (N=15):

- **Entry-point services:** Kubernetes auto-restart takes 75.1 s (detection delay + pod restart + stabilization). Agent-assisted recovery takes 73.4 s (13 s LLM inference + 0.1 s kernel + 60.3 s pod recovery). Improvement: ∼2%.
- **Non-entry-point services:** Kubernetes auto-restart takes ∼10 s (fast detection, quick restart). Agent-assisted recovery takes ∼23 s (13 s LLM overhead + 10 s recovery). The LLM inference time exceeds any benefit, making agent-assisted recovery **2.3× slower** than auto-restart for these services.

Agent-assisted remediation provides marginal TTR improvement for entry-point services where detection delays dominate, but degrades TTR by 2.3× for services with fast auto-restart. Typed actuation is primarily a *safety* mechanism, not a speed optimization. The 13 s LLM overhead is not a fundamental limit: we did not optimize for inference latency, and techniques such as prefix caching, smaller distilled models deployed near the infrastructure, or warm-started inference could reduce it substantially, narrowing or eliminating the TTR gap for non-entry-point services.

**TTR decomposition.** The median 43.5 s TTR for ISA+Critic breaks down as: diagnosis (5.2 s), planning (4.8 s), verification (3.1 s), kernel execution (0.1 s), and pod recovery (30.3 s). LLM inference (diagnosis + planning + verification = 13.1 s) accounts for 30% of TTR; actual infrastructure recovery (pod restart + stabilization) accounts for 70%. Microkernel overhead is negligible (<0.3%).

**Reproducibility.** Across three runs with different random seeds (N=30 each, 90 total), we observed 0% harm and 100% plan approval. The zero variance reflects deterministic ISA constraints and conservative agent policies; the planner repaired all rejected plans within three retries.

**Parallel execution.** For incidents affecting multiple services, the microkernel executes non-conflicting actions in parallel using conflict-key analysis, serializing only actions targeting the same resource. In micro-benchmarks (100 ms per-action latency, sufficient cluster resources), parallel execution provides **5× speedup**: 510 ms sequential → 102 ms parallel for 5 non-conflicting actions. Operators can configure `MAX_BATCH_SIZE` based on cluster capacity.

**Answer to RQ3.** *Partially.* For single-service incidents, agent-assisted remediation provides marginal TTR improvement (∼2%) for entry-point services where detection delays dominate, but *increases* TTR by 2.3× for services with fast auto-restart due to LLM inference overhead. For multi-service incidents, parallel execution provides up to 5× speedup. Speed gains are scenario-dependent; the primary contribution is safety (95% harm reduction).

### 6.5 RQ4: Generalization across Fault Types

We evaluate generalization using Chaos Mesh to inject five fault types (N=20 each on Social Network): pod failure, network partition, CPU stress, memory stress, and I/O delay. ISA-constrained remediation achieved 0% harm across all fault types (100 incidents total). We also validated on Hotel Reservation with pod failure (N=10) and network partition (N=5), achieving 0% harm. Typed actions allow agents to express fault-appropriate remediation (Restart, Drain+Restart, Scale+Restart, CircuitBreak+Restart) while the microkernel preserves safety.

**Answer to RQ4.** *Yes.* The system generalizes across diverse fault types (5 types) and workloads (2 DeathStarBench applications) with 0% harm, validating that typed actuation enables safe remediation without fault-specific logic.

## 7 Related Work

**Microreboot and crash-only software.** Crash-only software [3] established restart as a primary recovery mechanism. Microreboot [4] introduced recovery groups for co-dependent components. In their J2EE setting, static deployment descriptors (Enterprise JavaBeans dependencies) determined these boundaries. Our work extends this to microservice systems where dependency structure is dynamic and changes with feature flags, canary rollouts, and traffic shifting. The central difference is that we infer recovery scope *online* from distributed traces, reflecting the current workload rather than static configuration. We also generalize the actuation primitive beyond restart to include traffic management and resource adjustment.

**Table 4.** Comparison of safe actuation approaches.

| **Approach** | **Flexibility** | **Safety Mechanism** | **Rollback** |
|---|---|---|---|
| Raw tools (`kubectl`, APIs) | High | None | Manual |
| Policy verification | High | Post-hoc validation | Tool-specific |
| Undo mechanisms | High | State snapshots | Automatic |
| **Typed ISA (ours)** | Constrained | Effect-type enforcement | Transactional |

**Safe agent execution.** Recent systems explore safeguards for agent actions. STRATUS [7] formalizes "transactional no-regression" and implements undo-and-retry for agentic site reliability engineering (SRE). VeriGuard [19] intercepts and verifies agent actions against policies. Our approach differs in *when* the system enforces safety: typed actuation constrains what agents can *express* (pre-execution), while policy verification and undo validate or revert what agents *did* (post-hoc). Pre-execution constraints reduce policy authoring burden (agents cannot express actions outside the ISA) but sacrifice flexibility. STRATUS-style undo preserves flexibility but requires state snapshots. Table 4 summarizes tradeoffs; in practice, these approaches could be combined. Direct empirical comparison would require reimplementing policy languages, authoring equivalent policies, and controlling for policy completeness, which remains an open question. We compare tradeoffs rather than benchmarks. Our propose-validate-repair loop also draws on multi-agent error correction patterns; SQL-of-Thought [6] applies a similar iterative refinement strategy to reduce errors in text-to-SQL generation.

**Distributed tracing.** Dapper [26] demonstrated tracing at scale. We use modern tracing [22] to infer scope and ordering. Unlike observability, we compute enforceable safety boundaries.

**Benchmarks and fault injection.** DeathStarBench [11] provides representative microservice applications for end-to-end evaluation. Our online validation uses Chaos Mesh [5] to inject failures and measure recovery behavior, connecting actuation safety to realistic microservice failure modes.

**Compensating transactions.** Sagas [12] introduced compensation for long-running distributed operations. Our microkernel applies this pattern: actions are reversible or require explicit compensation for recovery.

**Service mesh and circuit breaking.** Service meshes (Istio [15], Linkerd [16]) provide circuit breaking and rate limiting at the infrastructure level. These mechanisms are orthogonal: service mesh policies define steady-state resilience, while our ISA enables active remediation during incidents. The `CircuitBreak` and `RateLimit` actions in our ISA can be implemented via service mesh APIs.

**Runbook automation and AIOps.** Traditional runbook automation encodes fixed remediation procedures; AIOps platforms extend this with anomaly detection and automated diagnosis. Our contribution is orthogonal: we provide a safe actuation layer that runbooks or AIOps systems can target. The typed ISA could serve as the actuation backend for existing platforms.

## 8 Limitations and Future Work

**Recovery speed.** Agent-assisted remediation provides modest TTR improvement for single-service failures (∼2%) and up to 5× speedup for multi-service incidents. For services with fast auto-restart, LLM overhead (∼13 s) exceeds baseline by 2.3× (∼23 s vs. ∼10 s). The primary value is *safety*, not speed.

**Harm estimation.** Our largest harm reduction (95%) comes from simulation with a harm model we calibrated to agent behavior. Online experiments validate that ISA constraints avoid regressions; aligning simulation with production incidents needs more work.

**External side effects.** Actions with external side effects (e.g., payments or user-visible messages) are treated as irreversible, requiring break-glass approval. Richer compensation strategies are left to future work.

**Trace quality.** Recovery-group inference depends on trace completeness [26] and consistency [14]. At 50% sampling, only 46% of edges are visible; missing traces can under-scope groups, the more dangerous failure mode. The conservative `MAX_GROUP_SIZE` cap and mandatory drain for high-connectivity services mitigate this. Retroactive tracing [29] could improve edge-case coverage.

**Scope and scale.** Our implementation assumes a single Kubernetes cluster; multi-cluster deployments need federated tracing. Online evaluation uses two DeathStarBench applications at moderate load; offline analysis covers 5,459 services, but we have not validated at high traffic.

## 9 Conclusion

Microreboot is unsafe in microservice systems where dependencies are dense, dynamic, and actuated by autonomous agents. We re-enable it by separating planning from actuation: a typed ISA with explicit effect semantics, a transactional microkernel, and trace-derived recovery boundaries. Evaluation on industrial traces and runnable workloads shows that we can infer recovery boundaries online with low latency, and typed actuation prevents harmful actions.

**Acknowledgments.** I thank the anonymous reviewers for their constructive feedback. I also thank all the people, in particular at MIT and MPI-SWS, with whom I have had insightful discussions on or adjacent to this topic over the past few years. I am grateful to Jean-Philippe Martin-Flatin for his mentorship and for co-founding a cloud service troubleshooting startup with me nearly fourteen years ago; the venture was ahead of its time, but the experience shaped my thinking on automated remediation. Large language models now make practical what we could only envision then. I also owe a debt to George Candea, whose teaching first grounded me in software engineering and whose original microreboot work inspired this paper.

**Disclosure of Interests.** The author has no competing interests to declare that are relevant to the content of this article.

## References

1. Alibaba Group: Alibaba cluster trace program: cluster-trace-microservices-v2021 (2021), `https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021`, accessed: 2024-01-15
2. Anand, V., Garg, D., Kaufmann, A., Mace, J.: Blueprint: A toolchain for highly-reconfigurable microservice applications. In: Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP 2023). pp. 482–497. ACM (2023). https://doi.org/10.1145/3600006.3613138, `https://doi.org/10.1145/3600006.3613138`
3. Candea, G., Fox, A.: Crash-only software. In: Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX). pp. 67–72. USENIX Association (2003), `https://www.usenix.org/legacy/event/hotos03/tech/full_papers/candea/candea.pdf`
4. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot—a technique for cheap recovery. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004). pp. 31–44. USENIX Association (2004), `https://www.usenix.org/legacy/event/osdi04/tech/full_papers/candea/candea.pdf`
5. Chaos Mesh Authors: Chaos Mesh: A powerful chaos engineering platform for kubernetes (2024), `https://chaos-mesh.org`, official project website
6. Chaturvedi, S., Chadha, A., Bindschaedler, L.: SQL-of-Thought: Multi-agentic text-to-SQL with guided error correction. In: Deep Learning for Code Workshop at the 38th Conference on Neural Information Processing Systems (DL4C @ NeurIPS 2025) (2025), `https://arxiv.org/abs/2509.00581`
7. Chen, Y., Pan, J., Clark, J., Su, Y., Zheutlin, N., Bhavya, B., Arora, R.R., Deng, Y., Jha, S., Xu, T.: STRATUS: A multi-agent system for autonomous reliability engineering of modern clouds. In: Advances in Neural Information Processing Systems (NeurIPS 2025) (2025), `https://arxiv.org/abs/2506.02009`
8. Clopper, C.J., Pearson, E.S.: The use of confidence or fiducial limits illustrated in the case of the binomial. Biometrika **26**(4), 404–413 (1934). https://doi.org/10.1093/biomet/26.4.404, `https://doi.org/10.1093/biomet/26.4.404`
9. Crume, A., Cepoi, A., Granados, C., Loza, R., McGhee, S., Gites, S., Mattson-Hamilton, T., Stacey, V.: Google site reliability engineering: Incident management guide. Google Site Reliability Engineering (nd), `https://sre.google/static/pdf/IncidentManagementGuide.pdf`, undated official PDF
10. Dean, J., Barroso, L.A.: The tail at scale. Communications of the ACM **56**(2), 74–80 (2013). https://doi.org/10.1145/2408776.2408794, `https://doi.org/10.1145/2408776.2408794`
11. 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., Espinosa, M., Lin, R., Liu, Z., Padilla, J., Delimitrou, C.: An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019). pp. 3–18. ACM (2019). https://doi.org/10.1145/3297858.3304013, `https://doi.org/10.1145/3297858.3304013`

12. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data. pp. 249–259. ACM (1987). https://doi.org/10.1145/38713.38742, `https://doi.org/10.1145/38713.38742`
13. Huang, L., Magnusson, M., Muralikrishna, A.B., Estyak, S., Isaacs, R., Aghayev, A., Zhu, T., Charapko, A.: Metastable failures in the wild. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). pp. 73–90. USENIX Association (2022), `https://www.usenix.org/conference/osdi22/presentation/huang-lexiang`
14. Huye, D., Liu, L., Sambasivan, R.R.: Systemizing and mitigating topological inconsistencies in alibaba's microservice call-graph datasets. In: Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE 2024). pp. 276–285. ACM (2024). https://doi.org/10.1145/3629526.3645043, `https://doi.org/10.1145/3629526.3645043`
15. Istio Project Authors: Istio: Connect, secure, control, and observe services (2024), `https://istio.io/`, open-source service mesh; CNCF graduated project. Accessed: 2024-01-15
16. Linkerd Project Authors: Linkerd: Ultralight, security-first service mesh for Kubernetes (2024), `https://linkerd.io/`, open-source service mesh; CNCF graduated project. Accessed: 2024-01-15
17. Meinicke, J., Wong, C.P., Vasilescu, B., Kästner, C.: Exploring differences and commonalities between feature flags and configuration options. In: Proceedings of the 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '20). pp. 233–242. Association for Computing Machinery (2020). https://doi.org/10.1145/3377813.3381366, `https://doi.org/10.1145/3377813.3381366`
18. Meta Platforms, Inc. and affiliates: Distributed tracing data from meta's microservices architecture (summary_data_atc23) (2023), `https://github.com/facebookresearch/distributed_traces`, accessed: 2024-01-15
19. Miculicich, L., Parmar, M., Palangi, H., Dvijotham, K.D., Montanari, M., Pfister, T., Le, L.T.: VeriGuard: Enhancing llm agent safety via verified code generation. arXiv preprint (2025). https://doi.org/10.48550/arXiv.2510.05156, `https://arxiv.org/abs/2510.05156`
20. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems **17**(1), 94–162 (1992). https://doi.org/10.1145/128765.128770, `https://doi.org/10.1145/128765.128770`
21. OpenAI: GPT-4 technical report. arXiv preprint (2023). https://doi.org/10.48550/arXiv.2303.08774, `https://arxiv.org/abs/2303.08774`
22. OpenTelemetry Authors: OpenTelemetry: High-quality, ubiquitous, and portable telemetry (2024), `https://opentelemetry.io/`, accessed: 2024-01-15
23. Rancher Labs: K3s: Lightweight Kubernetes (2024), `https://k3s.io/`, CNCF sandbox project. Accessed: 2024-01-15
24. Ruan, Y., Dong, H., Wang, A., Pitis, S., Zhou, Y., Ba, J., Dubois, Y., Maddison, C.J., Hashimoto, T.: Identifying the risks of LM agents with an LM-emulated sandbox. In: The Twelfth International Conference on Learning Representations (ICLR 2024) (2024), `https://arxiv.org/abs/2309.15817`
25. Schermann, G., Cito, J., Leitner, P., Zdun, U., Gall, H.C.: We're doing it live: A multi-method empirical study on continuous experimentation. Information and Software Technology **99**, 41–57 (2018).

https://doi.org/10.1016/j.infsof.2018.02.010, https://doi.org/10.1016/j.infsof.2018.02.010

26. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google (2010), https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/
27. The Kubernetes Authors: Liveness, readiness, and startup probes. Kubernetes Documentation (2025), https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/, last modified June 27, 2025
28. Ye, J., Li, S., Li, G., Huang, C., Gao, S., Wu, Y., Zhang, Q., Gui, T., Huang, X.: ToolSword: Unveiling safety issues of large language models in tool learning across three stages. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 2181–2211. Association for Computational Linguistics (2024). https://doi.org/10.18653/v1/2024.acl-long.119, https://aclanthology.org/2024.acl-long.119/
29. Zhang, L., Xie, Z., Anand, V., Vigfusson, Y., Mace, J.: The benefit of hindsight: Tracing edge-cases in distributed systems. In: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). pp. 321–339. USENIX Association (2023), https://www.usenix.org/conference/nsdi23/presentation/zhang-lei