International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I1P148
Eureka Vision Publication | Volume 7, Issue 1, 338-345 2026

Original Article

Adaptive Ephemeral Chaos Infrastructure for Resilience Validation in AWS Multi-Account Environments

Tripatjeet Singh
Senior Cloud Engineer Dallas-Fort Worth, USA.

Received On: 06/02/2026 ***Revised On: 07/03/2026*** ***Accepted On: 13/03/2026*** ***Published On: 19/03/2026***

Abstract - Running chaos engineering experiments safely across multiple AWS accounts is harder than it sounds. Most existing tools require manual setup per account, use fixed experiment templates that do not adapt to the application, and have no reliable way to confirm the system actually recovered after a test. This paper presents Adaptive Ephemeral Chaos Infrastructure (ECI), a framework that addresses these problems through four specified capabilities: a seven-phase experiment lifecycle that enforces safety at every step, a convergent recovery verification protocol that confirms system stability before an experiment is declared complete, a CI/CD integration with distributed collision detection, and a multi-account coordination protocol that synchronizes fault injection across AWS accounts. ECI also proposes two concepts reserved for future implementation i.e., an Application Dependency Graph for automatic topology mapping, and a centrality-based adaptive experiment selection model, described here at a design level to establish the intended direction of the framework. The paper concludes with a proposed evaluation methodology that outlines how the four implemented capabilities would be measured in a structured test environment.

Keywords - Chaos Engineering; AWS Fault Injection Simulator; Cloud Resilience; Distributed Systems; Multi-Account Architecture; CI/CD Orchestration; Convergent Restoration; Ephemeral Infrastructure.

1. Introduction

Most enterprise cloud applications today span several AWS accounts e.g. production, development, QA, and often more. Within each account, applications use containers, Lambda functions, managed databases, queues, and event-driven services wired together in ways that are rarely simple. This works well for scalability and cost control, but it creates a testing gap: when things go wrong in production, it is usually not one service failing cleanly. It is a chain of partial failures, slow dependencies, and cascading timeouts that no unit test or load test ever simulated [1].

Chaos engineering fills this gap by deliberately injecting real faults which is stopping containers, throttling databases, filling queues and observing how the system holds up. Netflix made the approach famous [2], and it is now the recognized standard for finding resilience problems before they reach users. The Site Reliability Engineering discipline reinforces the same point, that systems must be tested under realistic failure conditions as a routine operational practice, not as a one-off exercise [3].

Applying it safely at enterprise scale, however, is harder than running a single experiment. Three specific problems motivated this work. First, existing tools such as AWS FIS [4], Gremlin [5], Chaos Mesh [6], LitmusChaos [7] require engineers to manually write experiment templates for each service. As applications change with every deployment, those templates go stale quickly. Second, coordinating experiments across multiple AWS accounts has no standard mechanism; every tool we evaluated required per-account manual configuration. Third, none of these tools formally verify recovery. They stop the fault injection process, but whether the application actually returned to normal is left to the engineer to judge by looking at dashboards.

This paper presents Adaptive Ephemeral Chaos Infrastructure (ECI), a framework designed for AWS multi-account environments. ECI addresses all three problems through four specified and architecturally defined capabilities: a structured seven-phase experiment lifecycle, a convergent recovery verification protocol, a CI/CD pipeline integration with collision detection, and a multi-account coordination mechanism.

Additionally, this paper introduces two concepts that represent the intended future direction of ECI: an Application Dependency Graph (ADG) that would automatically map application topology from AWS metadata, and a centrality-based adaptive experiment selection model that would use that map to intelligently sequence experiments. These are presented as design proposals only. Neither is implemented in the current version of ECI. Full specification and implementation of both are reserved for the follow-on paper in this series.

Table 1 summarizes where ECI stands today compared to existing tools, using a two-tier distinction between capabilities that are implemented and those that are proposed.

2. Background and Related Work

The practice of deliberately breaking systems to find resilience gaps goes back to Netflix's Chaos Monkey [2],

which randomly terminated production EC2 instances. The Principles of Chaos Engineering [8] later formalized the methodology: define a steady-state hypothesis, vary something real, run the experiment, and look for deviation. That simple loop is still the foundation of everything built since.

The academic study of chaos engineering as an engineering discipline is more recent. Basiri et al. described Netflix's Chaos Automation Platform in 2019 [9], which automated the execution of chaos experiments in production and is the closest academic precedent to what ECI proposes. The same group published the canonical IEEE Software definition of chaos engineering [10], which remains the standard reference for the field's core principles. These works establish the production-engineering context that motivates ECI's design, though neither addresses multi-account orchestration or formal recovery verification.

Among commercial and open-source platforms, Gremlin [5] is mature and has solid rollback support, but it requires separate account configuration for each AWS environment and offers no topology-aware experiment selection. AWS FIS [4] is a well-built execution engine that handles the actual fault injection cleanly, but it is only an execution engine, it does not decide what to run, does not coordinate across accounts, and does not verify recovery. Chaos Mesh [6] and LitmusChaos [7] are excellent for Kubernetes clusters but do not cover AWS-managed services like Aurora, DynamoDB, or SQS. Steadybit [11] and Harness Chaos Engineering [12] come closer to enterprise integration but still use static experiment templates and offer no formal multi-account coordination.

AWS has also published a reference architecture for running chaos experiments across multiple accounts using FIS and AWS Control Tower [13]. It demonstrates that multi-account fault injection is feasible at the infrastructure level, but it describes a manual configuration process built on CloudFormation templates. It does not address lifecycle enforcement across experiment phases, CI/CD pipeline integration with collision detection, or metric-based recovery verification; the gaps that ECI is designed to close.

Table 1 compares these platforms against ECI, including the AWS multi-account reference architecture [13]. The ECI columns distinguish between what is implemented today and what is proposed for future work.

Table 1. Capability Comparison Across Chaos Engineering Platforms

Capability	FIS	Gremlin	C.Mesh	Litmus	Steadybit	Harness	ECI (This Paper)	ECI (Planned)
Multi-account coordination	No	No	No	No	No	No	Yes	—
CI/CD collision detection	No	No	No	No	No	No	Yes	—
Convergent recovery verification	No	No	Partial	Partial	No	No	Yes	—
Seven-phase experiment lifecycle	No	No	No	No	No	No	Yes	—
Org-level SCP guardrails	Partial	No	No	No	No	No	Yes	—
Planned Capabilities - Not Implemented in This Paper · Subject of Follow-On Work								
Topology-adaptive selection (ADG)	No	No	No	No	Partial	No	—	Planned
Centrality-based experiment ordering	No	No	No	No	No	No	—	Planned

Source: Rows above the divider reflect capabilities specified and architecturally defined in this paper. Rows below the divider (marked "Planned") reflect capabilities reserved for the follow-on paper and are not implemented or claimed in this work.

Note: "ECI (This Paper)" reflects capabilities that are architecturally specified in Sections III–VII. "ECI (Planned)" reflects design concepts introduced in Sections IV and VII that are reserved for implementation and empirical evaluation in the follow-on paper.

3. System Architecture

ECI is organized into three layers: a centralized control plane, the application accounts where faults are injected, and an observability layer that captures what happens during and after each experiment. Fig. 1 shows how these fit together.

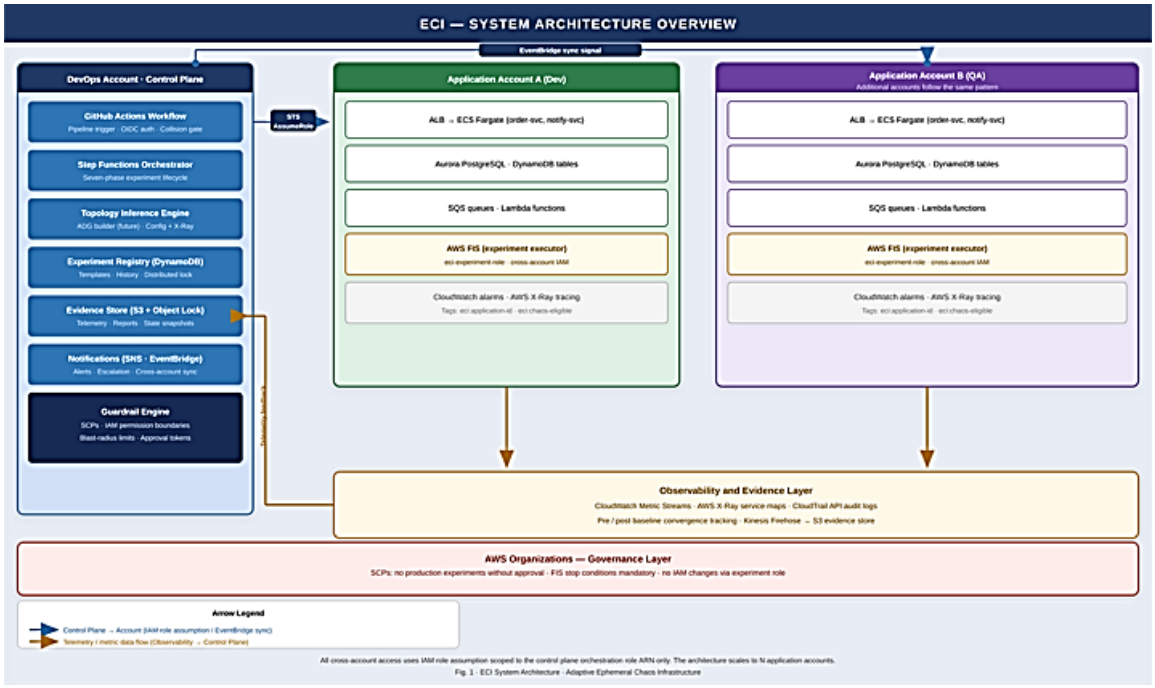

Figure 1. ECI System Architecture: Control Plane in the Devops Account, Application Accounts, and Shared Observability Layer

3.1. Control Plane

The control plane lives in a dedicated DevOps AWS account that contains no application workloads. Keeping it separate means, it cannot accidentally become a target of the experiments it manages. Five components sit here: the Step Functions workflow that runs the experiment lifecycle, the Lambda functions that handle orchestration logic, the DynamoDB tables that store experiment templates and history, the S3 bucket (with Object Lock) that holds all experiment evidence, and the SNS and EventBridge resources used for notifications and cross-account synchronization.

3.2. Application Accounts

The application accounts for development, QA, and others are where faults actually get injected. Each account exposes a cross-account IAM role that the control plane can assume, scoped narrowly to running FIS experiments and reading CloudWatch metrics. Resources eligible for chaos testing are identified by three AWS resource tags: eci:application-id, eci:criticality-tier, and eci:chaos-eligible. Anything not tagged is outside ECI's reach entirely.

3.3. Governance

AWS Organizations Service Control Policies enforce three rules at the organizational boundary: the experiment role cannot make IAM changes; FIS experiments cannot run in production-classified accounts without a manually signed approval token; and every FIS experiment must reference a CloudWatch alarm as a stop condition, preventing any experiment from running without an automatic cutoff. This governance model aligns with the reliability and security principles of the AWS Well-Architected Framework [14], specifically the practices of minimizing blast radius, enforcing least-privilege access, and automating operational safeguards.

4. Application Dependency Graph - Design Concept

One of the core limitations of current chaos engineering tools is that they treat every service the same. A fault injected into a peripheral cache is handled with the same tooling and the same risk model as a fault injected into the central database that every other service depends on. This creates real risk: a poorly chosen experiment sequence can cause a cascade that is hard to untangle and harder to attribute.

To address this, ECI's intended architecture includes an Application Dependency Graph (ADG) which is a map of the application built automatically from AWS metadata rather than maintained by hand. The idea is straightforward: AWS Config already knows which resources exist, AWS X-Ray already tracks which services are calling which, and the Resource Groups Tagging API can scope this to a specific application. Combining these three sources gives a reasonably complete picture of how services relate to each other without requiring any manual documentation from the engineering team.

Once that map exists, it becomes possible to reason about which services are structurally important. A service that sits at the center of many dependency paths, one that many others call through, is a high-risk experiment target. A service at the edge of the graph is a safer starting point. The ADG is intended to make this reasoning explicit and automatic rather than leaving it to the judgment of whoever is writing the experiment template.

The ADG also introduces a confidence signal. If X-Ray sampling is low on part of the application, some edges in the map may be missing. Rather than silently proceeding on an incomplete picture, ECI would surface a topology confidence score with every ADG snapshot, letting the experiment author know how much to trust the current map before committing to a blast radius.

The ADG is a design concept at this stage. It is not implemented in the current version of ECI. The full specification of how dependencies are inferred, how structural importance is measured, and how the confidence score is derived, is the primary subject of the follow-on paper in this series. Fig. 2 illustrates the intended concept.

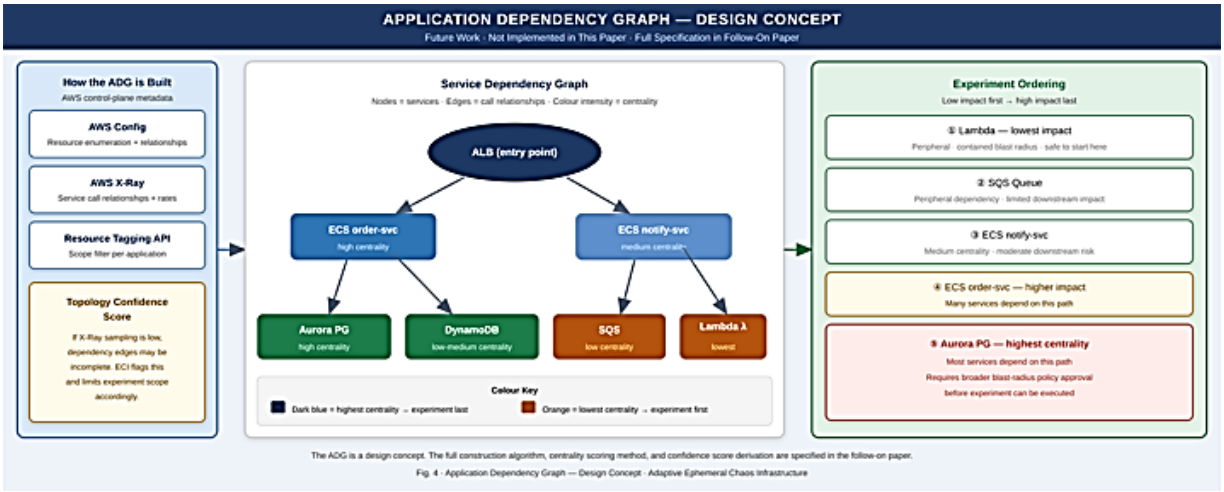

Figure 2. ADG Design Concept: AWS Metadata Sources Feed a Dependency Map Whose Structure Informs Experiment Ordering (Future Work)

5. Experiment Workflow

Every ECI experiment follows a fixed seven-phase sequence, implemented as an AWS Step Functions Express Workflow. The ordering is strict by design: phases cannot be skipped, fault injection cannot start until the safety phases are done, and the experiment cannot be declared complete until the system has been verified as recovered. Fig. 3 shows the full sequence.

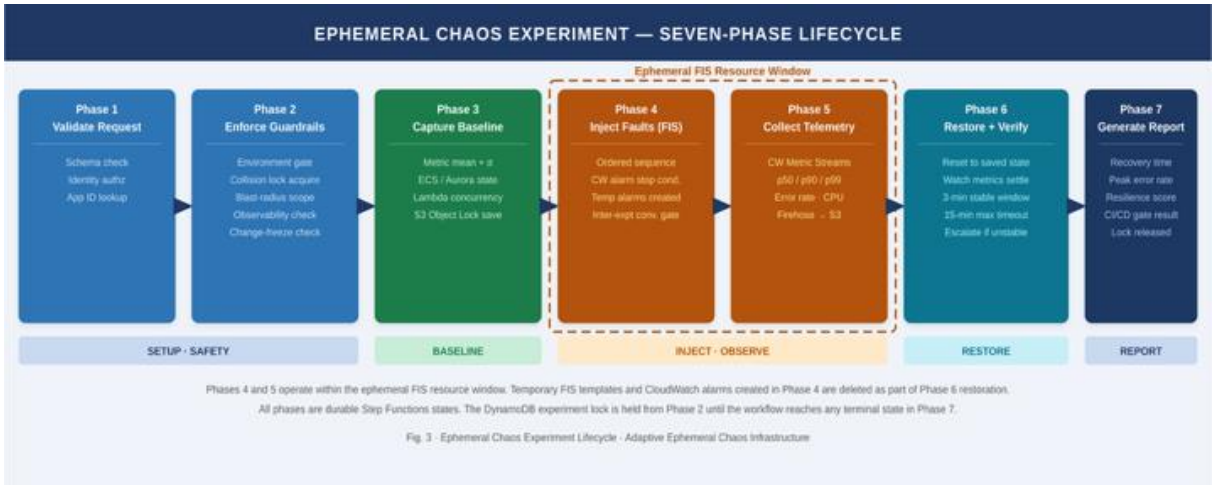

Figure 3. Seven-Phase Experiment Lifecycle. The Ephemeral FIS Window Covers Only Phases 4 And 5

5.1. Validation and Safety Checks

The first two phases handle validation and guardrail enforcement. Validation checks that the experiment request includes all required fields and that the person submitting it has permission to run experiments against the specified application. The guardrail phase then runs five checks in sequence: whether the target environment is cleared for chaos testing; whether another experiment is already running against the same services (enforced with a DynamoDB conditional write lock); whether the requested experiment scope is within the allowed blast-radius policy; whether the target services have enough CloudWatch metric history to establish a meaningful baseline; and whether any target resources were recently deployed (recently changed services are excluded because their metrics may not yet reflect stable normal behavior).

5.2. Baseline Snapshot

Before any fault is injected, ECI records the system's current normal state. This means capturing the mean and standard deviation of key metrics which is request latency, error rate, and resource utilization, over a 15-minute window, alongside configuration details like ECS service counts, Aurora cluster endpoints, and Lambda concurrency settings. The snapshot is stored in S3 with Object Lock enabled, making it tamper-evident and reliable as the reference point for both restoration and recovery verification later.

5.3. Fault Injection

Fault injection uses AWS FIS as the underlying execution engine. ECI builds FIS experiment templates dynamically from a catalog of supported fault types, targeting each service in the experiment plan in sequence. Each template includes a CloudWatch alarm as a stop condition; if the system degrades past the alarm threshold, the experiment stops automatically. Where a suitable alarm does not already exist for a target service, ECI creates a temporary one based on the baseline snapshot values and removes it after the experiment finishes.

When a test plan includes more than one experiment, a sequential safety rule applies: the next experiment only starts after the system has been verified as recovered from the previous one. This inter-experiment recovery gate prevents fault conditions from stacking up across a sequence of tests in a way that would make the results impossible to interpret.

5.4. Recovery Verification

After the FIS experiment has concluded, ECI does not immediately mark it as complete. It monitors the CloudWatch metrics and waits for all measured values to settle back within two standard deviations of their pre-experiment baseline for a continuous three-minute window. Only when that stability condition holds, is the experiment marked complete, and the system considered recovered.

If after 15 minutes the CloudWatch metrics show that nothing has stabilized ECI escalates and page the on-call Engineer via SNS. It also saves the current system configuration for investigation and marks the experiment as unresolved. This ensures that no subsequent experiment is triggered against a system that has not been confirmed stable, and that non-recovery events are surfaced to the team rather than silently logged.

6. CI/CD Integration

ECI runs inside regular CI/CD pipelines. Resilience testing becomes a quality gate in the same pipeline that deploys code, not a separate manual process. Fig. 4 shows the full flow from pipeline trigger to outcome.

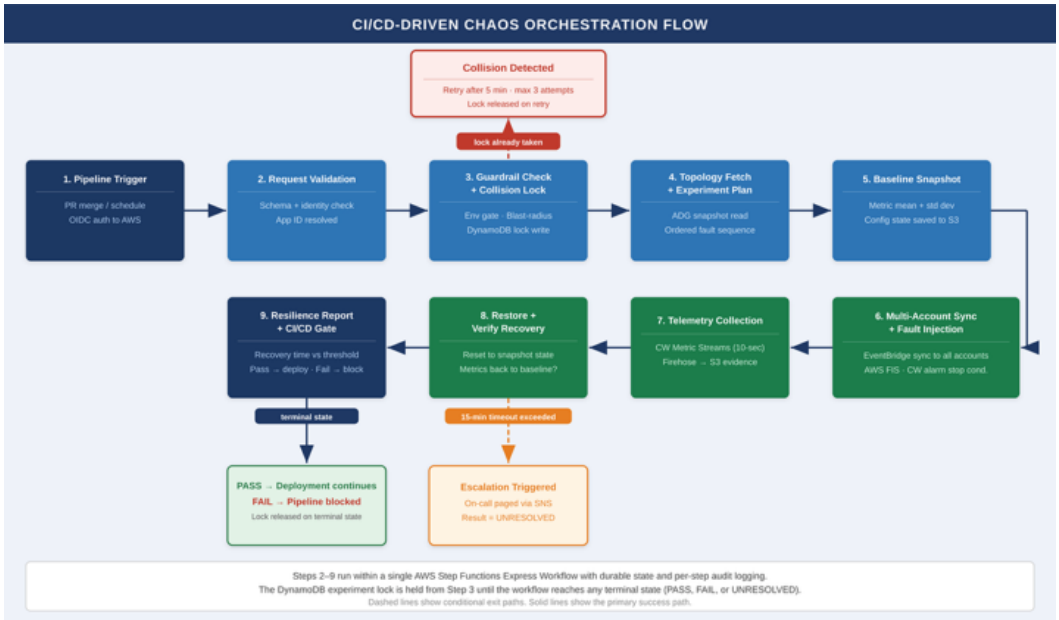

Figure 4. CI/CD Orchestration Flow: Nine Steps from Pipeline Trigger to Pass/Fail Gate, With Collision and Escalation Paths

6.1. Pipeline Integration

ECI is packaged as a reusable GitHub Actions composite action. When triggered by a pull request merge, workflow dispatch or a scheduled run, it authenticates to AWS using OpenID Connect federation, no long-lived credentials in repository secrets. The action submits an experiment request, polls the Step Functions execution until it finishes, and evaluates the result against a configurable recovery time threshold. If recovery time was within bounds, the pipeline passes, and deployment continues. If not, the pipeline fails and blocks the deployment.

6.2. Collision Detection

Running chaos experiments from CI/CD pipelines introduces a new risk: two pipeline runs triggered at nearly the same time can both attempt experiments against the same

services simultaneously. The results would be meaningless and the system state unpredictable. ECI prevents this with a distributed lock written to DynamoDB before any experiment starts. The write uses a conditional expression that only succeeds if no lock already exists for the same combination of application and target resources. If the lock is taken, the pipeline receives a conflict response and retries automatically after five minutes, up to three times. The lock releases as soon as the experiment reaches any terminal state.

6.3. Multi-Account Coordination

For experiments that span multiple AWS accounts, ECI uses a two-phase approach. First, it sends a prepare signal to all target accounts in parallel and waits for each to confirm its baseline snapshot is complete and it is ready to proceed. If any account cannot prepare, because of a guardrail failure, an observability gap, or a lock conflict, the whole experiment is cancelled before any fault is injected anywhere. Once all accounts confirm ready, ECI sends a synchronized start signal via EventBridge and all accounts begin their FIS experiments within seconds of each other. The primary source of inter-account timing variation is EventBridge delivery latency, which is independent of ECI's coordination logic and is bounded by AWS service-level characteristics for the target region.

7. Adaptive Experiment Selection - Design Concept

In the current version of ECI, experiment targets and sequences are specified manually by the engineer submitting the request. This works, but it relies on the engineer's knowledge of the application to make good choices. A service that looks unimportant in isolation might be a critical dependency for five other services. Without a systematic way to surface that kind of structural information, experiment planning stays a judgment call.

The intended solution is an adaptive selection model that builds on the ADG described in Section IV. Once the dependency map exists, ECI would assign each candidate service a score reflecting how much damage a fault there is likely to cause. Three factors would feed that score: how central the service is in the dependency graph, how critical it is according to its tag, and how often it has shown resilience issues in past experiments. Services with low scores would be targeted first, services with high scores last. The blast-radius policies i.e. isolated, correlated, or full, would then set the outer bound on how broadly the experiment spreads.

This model would remove the manual planning step for routine experiment runs and reduce the risk of accidentally starting with a high-impact target. It would also make ECI's experiment choices explainable: the score for each service would be visible in the experiment plan, so engineers could review and override the ordering if they had reasons to.

Like the ADG, this selection model is a design concept at this stage. It is not implemented, it was not used in the evaluation, and it depends on the ADG being built first. The scoring formula, algorithm design, and parameter choices are part of the scope of the follow-on paper.

8. Proposed Evaluation Methodology

ECI is an architectural framework specification at this stage. The capabilities described in Sections III through VII are fully designed but not yet empirically validated. This section defines the evaluation methodology that would be used to assess the four core capabilities in a structured test environment, providing a precise basis for that future work.

8.1. Target Environment

The intended evaluation environment consists of three AWS accounts within a single AWS Organizations hierarchy: one DevOps account hosting the ECI control plane, one development account, and one QA account. Both application accounts would host an identical synthetic order-processing application which would comprises an Application Load Balancer, two ECS Fargate services, an Aurora PostgreSQL cluster, two DynamoDB tables, two SQS queues, and three Lambda functions. This configuration provides ECI with an infrastructure that covers the complete range of AWS managed service types in the ECI Experiment Catalog; thus, it can be considered representative of an enterprise microservices application.

A synthetic load generator would maintain a constant, reproducible request rate against the ALB throughout all experiments. A constant load rate is critical for establishing an accurate baseline; otherwise, variable traffic will prevent differentiating latency increases due to actual faults from normal fluctuations caused by varying demand levels. The load level would be set to produce stable p99 latency and error rate metrics over a 15-minute baseline window that will serve as the statistical foundation for the convergence verification checks in Phase 6.

8.2. Capability-Specific Evaluation Criteria

Each of ECI's four core capabilities has a distinct measurable success criterion. The seven-phase lifecycle and guardrail enforcement would be assessed by submitting experiment requests that violate each guardrail condition in turn, targeting a production-classified account, attempting to acquire a lock already held, targeting a service with insufficient metric history, and targeting a recently deployed resource. Each should produce a clean rejection at the appropriate phase with no fault injection occurring. A secondary assessment would verify that all seven phases complete in the correct sequence for valid requests, with Step Functions execution history serving as the audit record.

Convergent recovery verification would be assessed across multiple experiment types by comparing the system's CloudWatch metric values before and after each experiment at the 3-minute stable window boundary. The primary success criterion is that ECI correctly identifies recovery, or correctly escalates non-recovery rather than making an incorrect binary determination. The escalation path specifically would be tested by injecting a fault whose recovery naturally exceeds the 15-minute timeout,

confirming that the result is marked UNRESOLVED and an SNS notification is dispatched rather than a false PASS being logged.

CI/CD collision detection would be evaluated by triggering two pipeline runs against the same application target within a narrow time window and confirming that the second run receives a COLLISION_DETECTED response rather than proceeding to fault injection. The lock release mechanism would be verified by confirming that a subsequent run succeeds after the first experiment reaches its terminal state.

Multi-account coordination would be assessed by running the Aurora failover experiment across both application accounts simultaneously and measuring the interval between the EventBridge sync signal and each account's FIS StartExperiment API call. The success criterion is that both accounts begin their experiments within a consistent, bounded window that falls within AWS EventBridge's documented delivery latency range for the target region.

8.3. What This Evaluation Does Not Cover

The proposed evaluation is deliberately scoped to the four capabilities this paper specifies. It does not cover the ADG construction or adaptive experiment selection, which are the subject of the follow-on paper and would require their own evaluation criteria. It does not cover production workload behavior, multi-region coordination, or the long-term learning properties of the experiment history store. Those are explicitly future work. Any future empirical study building on this methodology should report results with appropriate statistical rigor including repeated trials, confidence intervals, and clearly defined metric collection windows, rather than single-point measurements.

9. Limitations

The most significant limitation of the current version of ECI is the absence of topology-aware experiment selection. Experiment targets and sequences must be specified manually by the engineer submitting the request, which means the quality of a test plan depends entirely on that engineer's understanding of the application topology. The ADG and adaptive selection model in Sections IV and VII are designed to address this limitation, but they are not yet built.

The convergent recovery verification protocol relies on having sufficient CloudWatch metric history to establish a meaningful baseline. Applications with inadequate observability such as missing metrics, very low sampling rates, or freshly deployed services with no history, may not have enough signal to define a reliable baseline. ECI's guardrail phase is designed to detect and reject these cases before any experiment runs, but the underlying observability gap is an application-level problem that ECI cannot resolve on behalf of the team.

The recovery threshold parameters i.e. two standard deviations from baseline over a three-minute stable window, are reasonable defaults derived from common operational practice, but they have not been empirically optimized. Applications with inherently noisy metrics or naturally slow recovery patterns may need different settings. The follow-on paper's evaluation plan includes sensitivity analysis for these parameters.

The framework as specified targets single-region AWS deployments. Multi-region architectures introduce additional coordination complexity, particularly around inter-region EventBridge delivery latency and cross-region IAM trust; that is outside the current design scope and is identified as future work.

10. Future Work

The immediate next step for ECI is building the Application Dependency Graph. The metadata needed are resource relationships from AWS Config, call patterns from X-Ray, scope filters from the tagging API, which is already available in most AWS environments. The work is in turning that raw metadata into a reliable, queryable dependency map and validating that the map is accurate enough to trust for experiment planning. This is the primary subject of the follow-on paper.

Once the ADG exists, the adaptive selection model becomes buildable. The design described in Section VII needs a working graph to operate on. The two are deliberately sequenced: the follow-on paper specifies both, with the ADG construction coming first.

Two other improvements are on the roadmap but further out. One is learning from past experiments, using the history stored in the Experiment Registry to update the weight given to different risk factors over time, so ECI's selections improve as it accumulates data about which experiments actually find problems. The other is multi-region support, which requires solving a harder coordination problem given the additional latency and governance complexity involved.

11. Conclusion

ECI is a response to three practical gaps that organizations running chaos engineering at enterprise scale consistently encounter: no standard mechanism for coordinating experiments across multiple AWS accounts, no formal way to verify that a system has actually recovered after a fault rather than just assuming it has, and no safe handling of concurrent pipeline runs that could trigger overlapping experiments against shared infrastructure.

This paper has specified four architectural capabilities that address these gaps directly: a seven-phase experiment lifecycle with strictly enforced phase ordering, a convergent recovery verification protocol with a defined stability criterion and an escalation path for non-recovery events, a CI/CD integration using distributed locking to prevent experiment collisions, and a two-phase multi-account coordination protocol using EventBridge synchronization.

Each capability is described with enough architectural and behavioral specificity to serve as a basis for implementation and subsequent empirical validation.

Two further capabilities which is the Application Dependency Graph and the adaptive experiment selection model, are introduced as design concepts that define where ECI is heading. They are not specified for immediate implementation. The follow-on paper in this series takes them as its primary subject, providing full construction algorithms, scoring models, and an empirical evaluation of both the ADG and the adaptive selection logic against the orchestration framework specified here.

The contribution of this paper is coherent: a clearly bounded, governance-integrated chaos orchestration framework with an explicit roadmap for the intelligent, topology-aware layer that will be built on top of it.

References

[1] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages," Proc. ACM Symposium on Cloud Computing (SoCC), 2016. [Online]. Available: https://ucare.cs.uchicago.edu/pdf/socc16-cos.pdf

[2] Y. Izrailevsky and A. Tseitlin, "The Netflix Simian Army," Netflix Technology Blog, Jul. 2011. [Online]. Available: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116

[3] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Site Reliability Engineering: How Google Runs Production Systems, O'Reilly Media, 2016. [Online]. Available: https://sre.google/sre-book/table-of-contents/

[4] Amazon Web Services, "AWS Fault Injection Simulator User Guide," AWS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/fis/latest/userguide/what-is.html

[5] Gremlin Inc., "Gremlin Chaos Engineering Platform," 2024. [Online]. Available: https://www.gremlin.com/docs

[6] Chaos Mesh Authors, "Chaos Mesh: A Powerful Chaos Engineering Platform for Kubernetes," CNCF Project, 2021. [Online]. Available: https://chaos-mesh.org/docs/

[7] U. Mahapatra et al., "LitmusChaos: A Cloud-Native Chaos Engineering Framework," Proc. IEEE International Conference on Cloud Engineering (IC2E), 2022. [Online]. Available: https://litmuschaos.io/

[8] Principles of Chaos Engineering, "Principles of Chaos Engineering," last updated Mar. 2019. [Online]. Available: https://principlesofchaos.org/

[9] A. Basiri, L. Hochstein, N. Jones, and H. Tucker, "Automating Chaos Experiments in Production," Proc. IEEE/ACM 41st Int. Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 31–40, May 2019. DOI: 10.1109/ICSE-SEIP.2019.00012. [Online]. Available: https://arxiv.org/abs/1905.04648

[10] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos Engineering," IEEE Software, vol. 33, no. 3, pp. 35–41, May–Jun. 2016. DOI: 10.1109/MS.2016.60. [Online]. Available: https://arxiv.org/pdf/1702.05843

[11] Steadybit GmbH, "Steadybit Chaos Engineering Platform Documentation," 2024. [Online]. Available: https://docs.steadybit.com/

[12] Harness Inc., "Harness Chaos Engineering Documentation," Harness Developer Hub, 2024. [Online]. Available: https://developer.harness.io/docs/chaos-engineering/

[13] Amazon Web Services, "Chaos Engineering Leveraging AWS Fault Injection Simulator in a Multi-Account AWS Environment," AWS Cloud Operations Blog, Mar. 2022. [Online]. Available: https://aws.amazon.com/blogs/mt/chaos-engineering-leveraging-aws-fault-injection-simulator-in-a-multi-account-aws-environment/

[14] Amazon Web Services, "AWS Well-Architected Framework Reliability Pillar," AWS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/welcome.html