International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 7, Issue 4, PP 16-29, 2026
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V7I4P103

Original Article

Predictive Self-Healing of Financial Trading Microservices Using Spatiotemporal GNNs and DORA-Aligned Evidence Generation

Sandipkumar Uchdadiya
Senior Solution Engineer, Independent Researcher, USA.

Received On: 24/08/2026 ***Revised On: 22/09/2026*** ***Accepted On: 29/09/2026*** ***Published On: 04/10/2026***

Abstract: *Financial trading platforms are built from microservices whose failures rarely stay where they start. Degradation in one service travels along call and data dependencies until order handling breaches its service levels. Reactive recovery acts after the breach, and predictive models that watch services one at a time miss propagation that is visible only in the dependency structure. This paper presents a closed-loop resilience controller that integrates four elements. The first is a multi-view service graph that combines dependency, call and telemetry-derived causal-dependency views at the granularity of service instances. The second is a spatiotemporal graph neural network that predicts, for every instance, the probability of a service-level breach within a ten-minute horizon, together with the expected blast radius and the order of propagation. The third is a bounded remediation layer, in which a deterministic safety gate authorizes remediation tiers and enforces market-session constraints, and a supervised action-ranking policy trained from counterfactual replays of chaos scenarios selects among the permitted actions. The fourth is an evidence record designed to align with relevant provisions of the EU Digital Operational Resilience Act (DORA). The controller is evaluated in a simulation-based reference evaluation, a discrete-time simulation of a fourteen-service trading platform with chaos-injected faults, using five fault classes, five severity levels, three workload profiles and thirty repetitions, which gives 2,250 runs per method, split at the scenario level. On held-out test scenarios the proposed model predicted breaches with an F1 of 0.228 (95% CI [0.193, 0.260]) against a base rate of 0.026. It was ahead of a dynamic graph transformer on PR-AUC, of a temporal graph neural network on F1 and of a topology-free baseline on both, and statistically level with a per-instance recurrent model. Removing the recurrent layer reduced F1 by 0.028 (95% CI [0.018, 0.038]). The decisive component was action selection. With predictions held fixed, the learned policy reduced the blast radius by 1.63 instances (95% CI [0.83, 2.50]) and the mean time to recovery by 140 s (95% CI 75 to 213 s) relative to a rule-driven reactive policy, mostly by declining actions that the replays showed to be harmful. The full controller decided in 1.37 ms on average on a laptop CPU. The architecture produces technical evidence aligned with relevant DORA provisions; it does not establish regulatory compliance. All results come from simulation, and evaluation on benchmark deployments and live platforms remains future work.*

Keywords: *Spatiotemporal Graph Neural Networks, Self-Healing Systems, Failure Propagation, Microservices, Financial Trading Platforms, Chaos Engineering, AIOps, Causal Discovery, Operational Resilience, DORA, Abstention, Automated Remediation.*

1. Introduction

A trading platform fails in a particular way. A market-data adapter begins to degrade. Its own metrics move only slightly, but the pricing and order-routing services that depend on it will breach their service levels within minutes, and when they do, fills are affected. Self-healing in this setting means that the platform acts before a human does: it predicts from the dependency graph, not from a single metric, that pricing and order routing will breach within the next ten minutes, and it reroutes flow, drains the degraded instance and circuit-breaks the affected path before any fill is touched.

Ordinary reliability engineering stops at restoring service and minimizing the mean time to recovery (MTTR). Its tools are reactive: an alert fires when a latency percentile or an error rate crosses a threshold, and by then the incident already exists. Predictive approaches improve on this, but most of them model each service in isolation. A per-service model cannot see that a node whose own metrics look clean is about to fail because the queue of its upstream neighbor is building, and in a trading stack that dependency-mediated signal is where cascading failures begin.

Regulation adds a second reason to act early and to act accountably. The EU Digital Operational Resilience Act [1] has applied to financial entities since 17 January 2025. DORA requires financial entities to establish capabilities for

ICT risk management, protection, detection, response, recovery, and operational resilience testing. Predictive failure modeling can strengthen these capabilities by enabling mitigation before emerging degradation develops into a cascading ICT disruption. The regulation does not require prediction, and nothing in it names predictive or machine-learning methods. It requires resilience capabilities, and the architecture presented here is a way of operationalizing them, together with an evidence record of what the automated system did and why.

The research gap lies between four bodies of work that have advanced separately. Spatiotemporal graph learning predicts how signals evolve over a network [2], [3]. Microservice fault localization explains failures once they are observed [4], [5]. Self-healing and autonomic systems define the closed loop [6], [7]. Regulatory resilience frameworks define what an accountable response must leave behind [1]. A controller for regulated trading infrastructure needs all four at once, and their integration is the subject of this paper. The novelty of the proposed framework lies not in introducing a new graph-attention operator, but in integrating multi-view spatiotemporal failure-propagation modeling, bounded cost-sensitive remediation, and DORA-aligned evidence generation into a unified closed-loop resilience controller.

The author's earlier work addressed SLA-aware resource allocation on financial trading platforms with deep reinforcement learning [8]. The present paper addresses an adjacent problem on the same class of infrastructure. It predicts how failures propagate and bounds the automated response, its graph model is temporal and predictive rather than a structural state encoder, and its regulatory focus is DORA's detection, incident management and testing provisions.

1.1. Scope and Evidence

The architecture and its operating constraints come from the author's experience of trading infrastructure: the failure taxonomy, the dependency patterns that cascade, and the market-session rules that remediation must respect. The evidence comes from a reference implementation and a discrete-time simulation, built so that every number in this paper can be regenerated by any reader from the released code with fixed seeds. The simulation is a model of a generic trading platform, not a production system and not a deployment of a public microservice benchmark. Its findings are therefore claims about the architecture under the simulated dynamics, and Section 7 sets out what they do not establish.

1.2. Contributions

- A multi-view service graph at instance granularity, with a telemetry-derived causal-dependency view whose construction is specified for reproduction (Section 3.1).
- A spatiotemporal graph neural network with heads for breach probability, blast radius and propagation order over a ten-minute horizon, trained from chaos-generated labels that include near-miss negatives (Sections 3.2 and 3.3).
- A remediation design that keeps safety deterministic and learns only the ranking of permitted actions, trained by supervised learning from counterfactual replays of controlled chaos scenarios (Section 3.4).
- An abstention mechanism and an evidence record aligned with relevant DORA provisions (Sections 3.5 and 3.6).
- A reproducible evaluation that separates prediction from recovery, isolates the contribution of each component through ablation, compares predictors under the same action policy, and reports every result with a 95 percent confidence interval (Sections 4 and 5).

2. Related Work

2.1. Graph-Based Failure Prediction

Spatiotemporal graph neural networks combine graph convolution over a fixed topology with a temporal model, an approach established for traffic forecasting by STGCN [2] and T-GCN [3]. Dynamic graphs, whose topology itself changes, are handled by models such as EvolveGCN [9], which evolves the network parameters over time, and by dynamic graph transformers such as DyGFormer [10]. The building blocks used in this paper are standard: graph attention [11], relation-specific message passing for typed edges [12], graph convolution over a normalized adjacency [13], and gated recurrent units for temporal state [14]. Within microservices, Seer [15] showed that QoS violations can be predicted ahead of time from distributed traces, which established that the lead time needed for mitigation is available in the telemetry.

2.2. Microservice Fault Localization

Root-cause localization ranks candidate faulty services after an anomaly is observed. MicroRCA [4] localizes performance issues over an attributed service graph, Eadro [16] combines logs, metrics and traces in an end-to-end troubleshooting model, DejaVu [17] targets recurring failures with interpretable localization, and Sage [18] uses causal models over the service graph for performance debugging. An industrial study built around the TrainTicket benchmark characterized how faults in microservice systems manifest and are debugged [19], and a survey covers anomaly detection and root-cause analysis for service-based cloud applications [5]. Localization answers where a failure is. Prediction answers where it is going, which is the question remediation has to act on.

2.3. AI-Driven Self-Healing

The autonomic computing vision framed self-healing as a closed loop of monitoring, analysis, planning and execution [6], and a survey of self-healing systems catalogs the approaches that followed [7]. Large language models have recently been applied to incident management, recommending root causes and mitigation steps for cloud incidents [20], and automating root-cause analysis with retrieval of incident history [21]. Their latency and cost depend on model size, hardware and prompt length. This

paper does not evaluate an LLM-based controller, and it makes no comparative claim about one.

2.4. AIOps and Causal Telemetry

AIOps methods for failure management are surveyed in [22], and deep graph anomaly detection in [23]. Causal discovery from telemetry rests on constraint-based search, of which the PC algorithm is the canonical form [24]. PCMCI extends that search to high-dimensional time series with lagged dependencies and controls false positives through conditioning [25]. CauseInfer applied causality graphs to performance diagnosis in distributed systems [26]. Edges inferred this way describe statistical dependence under the assumptions of the method rather than proven causation, a distinction Section 3.1 carries into its terminology.

2.5. Chaos Engineering

Chaos engineering treats resilience as something to be tested experimentally by injecting faults into running systems [27], and its automation in production showed how such experiments can be run safely and repeatedly [28]. Public microservice benchmarks such as DeathStarBench [29] and TrainTicket [19] make fault-injection experiments on real deployments reproducible. This paper uses a simulation instead, which trades the realism of a deployment for exact control of every fault, seed and replay, and Section 7 returns to that trade.

2.6. DORA and Operational Resilience

DORA [1] organizes ICT resilience obligations around risk management, incident management, resilience testing and third-party risk. Article 8 requires identification of ICT assets and the mapping of their interdependencies, Article 10 requires mechanisms to promptly detect anomalous activities, and Article 17 requires an incident management process that includes early warning indicators. Articles 11 and 12 cover response and recovery, and backup, restoration and recovery procedures. Chapter IV, Articles 24 to 27, governs digital operational resilience testing, and Article 25(1) names scenario-based tests and end-to-end testing among the required types. The criteria for classifying incidents and the materiality thresholds for major incidents are specified in delegated regulation [30]. None of these provisions prescribes a method. They specify outcomes, which leaves room for, and does not require, the predictive approach described here.

2.7. Research Gap

Prediction, localization, remediation and regulatory evidence have each been studied, and largely in isolation. Graph-based predictors stop at a prediction, localization methods start after the failure, self-healing frameworks rarely bound their own actions against the cost of being wrong, and none of these lines of work produces the evidence a supervised financial entity has to keep. The gap is a controller that predicts propagation over the dependency structure, acts only within deterministic safety bounds, learns which permitted action works best, and records every decision, including the decisions not to act.

3. Methodology

Figure 1 shows the controller as a closed loop. Each stage consumes the output of the previous one, and every stage writes to the evidence record.

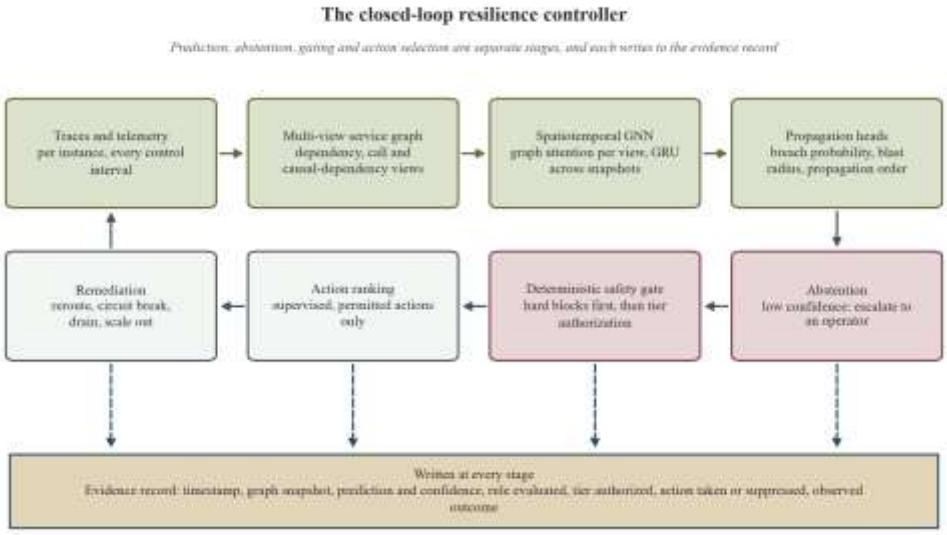


Fig 1: The Closed-Loop Resilience Controller. Prediction, Abstention, Gating and Action Selection are Separate Stages, and Each One Writes to the Evidence Record

3.1. Multi-View Graph Construction

Nodes are service instances, not service types. Failures are usually instance-local, and aggregating instances to their service type hides the signal. Each node carries a telemetry feature vector of nine values aggregated over a 60 s window: p50 and p99 latency, error rate, saturation, queue depth, retry count, timeout count, connection-pool utilization and garbage-collection pause time. Latencies, counts, queue depth and pause time are log-transformed, and every feature

is normalized against the healthy windows of its own service type.

The graph has three views and seven typed relations, shown in Figure 2. Edges carry degradation from callee to caller, the direction in which it propagates.

The *dependency view* records every declared caller and callee pair, including dependencies that carry no traffic. It describes what could talk to what, and it has three edge types: synchronous RPC, asynchronous queue and datastore access.

The *call view* records the pairs that carried at least one call in the window, with the same three edge types and two edge attributes, call volume and latency contribution. A registry-derived graph includes dependencies that carry no traffic and misses those that appear only under load, which are precisely the ones that cascade. The call view exists because of that difference.

The *telemetry-derived causal-dependency view* records lagged statistical dependence between instances. Each instance is summarized by a composite health score, the mean of the z-scores of its log latencies, queue depth, pool utilization and timeouts over a trailing block of 30 windows. A constraint-based search in the style of the PC algorithm [24], applied to lags of one to three windows in the manner of the first stage of PCMCI [25], retains an edge from instance i to instance j when a lagged score of i remains dependent on the score of j after conditioning on up to two other candidate parents, as judged by a two-sided Fisher z-test of partial correlation at a significance level of 0.01. Candidate parents are restricted to instances within two hops of j in the dependency view, the view is refreshed every 10 windows, and an edge is kept only if it appears in two consecutive refreshes. The result is described as a telemetry-derived causal-dependency graph, not as a true causal graph, because hidden common causes and the assumptions of the test can produce edges that do not correspond to causal mechanisms.

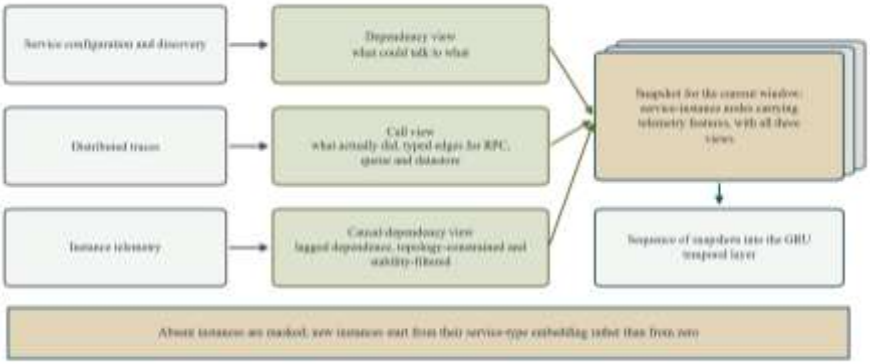


Fig 2: Multi-View Graph Construction. Each Window Produces a Snapshot with Three Views, and the Sequence of Snapshots is the Input to the Temporal Model

The graph is rebuilt every window rather than maintained as a persistent structure. The sequence of snapshots is the input to the temporal layer, so a change in topology is an input to the model rather than a disruption of it. Instances that appear and disappear through autoscaling, replacement and rolling restarts are normal traffic. Absent nodes receive no messages and no loss, and a new instance starts from a state derived from the learned embedding of its service type rather than from zero.

3.2. Spatiotemporal Graph Neural Network

Each snapshot passes through one relation-specific graph-attention layer [11], [12]. For each of the seven relations, the model has its own value projection and two attention heads, scores each in-edge from the endpoint states with a bias from the edge attributes, and normalizes the scores with a softmax over the node's in-edges of that relation. The messages of the seven relations are summed and fused with the node's own projection. A gated recurrent unit [14] then carries each node's state from one window to the next, so the representation of an instance reflects both its neighborhood and its recent history, and information travels one hop further with every window. The hidden size is 32 throughout, and the full model has 20,543 parameters (Table 2). None of these operators is new; Section 1 states where the contribution lies.

3.3. Failure Propagation Heads

The model has three output heads. The *breach head* gives, for each instance, the probability of a service-level breach within the prediction horizon H. The *blast-radius head* gives the probability that the instance belongs to the set that breaches after a fault. The *propagation-order head* gives a score that ranks instances by the order in which they are expected to breach. A breach is defined operationally rather than as a generic anomaly: p99 latency above the service's threshold, or error rate above its tolerance, sustained for three consecutive windows so that single spikes do not count. Thresholds are fixed before any model is trained, at twice the 99th percentile of fault-free p99 under the highest workload, with a one percent error tolerance.

The horizon is ten minutes, chosen backward from the cost of remediation. A prediction is useful only if it arrives

before the slowest remediation it is meant to enable can complete. Circuit-breaking and rerouting to warm capacity take seconds. Draining an instance and bringing a replacement to serving state, with scheduling, warm-up, connection pools and caches, takes two to five minutes in a trading stack. Rolling back a deployment or provisioning new capacity takes five to fifteen. Ten minutes covers that action space with margin. Below about two minutes, the only remediation still available is the one a threshold alert could have triggered anyway.

The signal comes from leading indicators and from structure. Error rate and p99 latency are lagging: by the time they move, the incident exists. Saturation, queue depth and pool utilization degrade first, and the simulation reproduces that ordering. At an instance hit by resource saturation, saturation departed from its baseline by more than four standard deviations an average of 4.2 windows before the breach and queue depth 3.8 windows before it, against 0.4 windows for error rate; under pool exhaustion, pool utilization led by 4.2 windows. The structural half of the signal is that a node's own metrics may look clean while its upstream neighbor's queue is building, and message passing over the graph is what turns that into a prediction.

Labels come from controlled fault injection, which inverts the rarity problem instead of working around it. Under injection, the fault, its target and its start time are controlled, so onset is exact and causal attribution is known. A node-window is labeled positive when the node's breach begins within the following H windows, so the model learns to fire across the whole usable lead time. Blast-radius labels come from the same runs.

Negatives include near-misses, injected degradations that did not breach; 1,118 of the 2,250 runs ended without any breach. Without them a model learns that any anomaly precedes a breach, which scores well and is operationally useless. Positive node-windows are about two percent of the training data, so training uses class-weighted focal loss [31], and the blast-radius and order heads add a focal loss and a pairwise ranking loss. Accuracy is never reported. The primary metrics are F1 at an operating threshold chosen on validation data and the area under the precision-recall curve (PR-AUC), both read against the base rate [32], and an always-negative predictor appears in the prediction table as the floor.

3.4. Action Policy Training

Remediation has two layers, and keeping them separate is the central design decision. A deterministic safety gate decides which actions are permitted. A learned policy ranks only the permitted actions.

The safety gate: Actions are tiered by cost and reversibility. Tier 1 is passive and always available: pre-warm replacement capacity and escalate to an operator. Tier 2 acts on traffic and is reversible in seconds: shift routing weights away from the degrading instance, trip a circuit breaker, shed non-critical load, or defer part of the instance's incoming work for later replay. Tier 3 acts on the lifecycle of instances and is slow to undo: scale a service out, or drain an instance and replace it. Rollback and state reconciliation belong to the design but have no counterpart in the simulation, because the fault design contains no faulty deployments. The gate authorizes a tier from the confidence of the prediction for the focus instance, the number of instances triggered and whether any of them lies on the order path. Tier 3 requires a confidence of at least 0.70, at least two triggered instances and an order-path instance among them. Tier 2 requires a confidence of at least 0.70, or at least 0.50 with two or more triggered instances. Tier 1 requires 0.30.

Market-session constraints are hard blocks, evaluated before any tier is considered. Stateful services on the live order path, the matching engine, order book and position keeper, are never acted on automatically: on those instances the controller pre-warms capacity, escalates and stops, because the cost of being wrong exceeds the cost of the predicted failure. Tiers 2 and 3 are suppressed across the opening and closing auctions and during volatility halts. Actions are capped at two per service and six per run, with a cooldown of three windows per instance, so that a mispredicting model cannot cascade its own remediation. Figure 3 summarizes the gate.

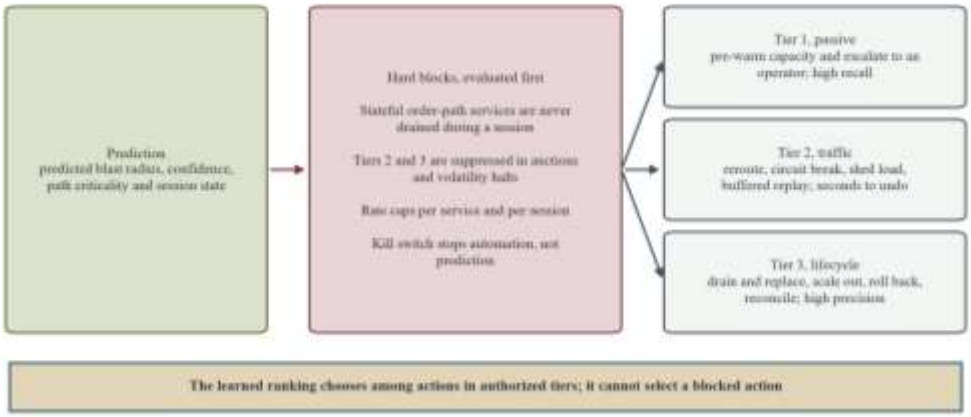


Fig 3: The Deterministic Safety Gate. Hard Blocks are Evaluated First, the Tier Is Authorized from the Prediction, and the Learned Policy Ranks Only What the Gate Permits

The learned policy: Within the permitted set, the choice of action is learned from counterfactual replays, as Figure 4 shows. On the training runs, the ST-GNN with the reactive policy (below) was run without acting, and the first two decisions per run at which the gate authorized Tier 2 or above were recorded, giving 1,626 decision situations. Each situation was replayed from the identical seeded state once per permitted action, 9,571 replays in all, with that single action applied and no other. Because the simulation's noise is keyed to the seed, every replay of a situation shares exactly the same randomness, so the difference between replays is the effect of the action alone. Each action a in situation v is scored by a utility

$$U(a, v) = \Delta\text{Containment}(a, v) - \lambda 1 \times \text{BlastCost}(a, v) - \lambda 2 \times \text{OperationalCost}(a, v)$$

where ΔContainment is the containment of the replay minus that of the Tier 1 replay, BlastCost is the number of instances that breached after the decision, OperationalCost is the share of entry requests shed, failed fast or deferred plus the capacity added or removed, and $\lambda 1 = 0.1$ and $\lambda 2 = 1.0$. Recovery time is recorded but not part of U. An action is excluded as unsafe if it caused a stateful order-path instance to breach that did not breach under Tier 1 alone; 140 replays were excluded on that ground. The best safe action becomes the training target,

$$a^* = \text{argmax over safe actions } a \text{ of } U(a, v)$$

And the action-selection head, a small network over the ST-GNN embeddings of the focus and the triggered set, the prediction statistics and the focus telemetry, is trained as a supervised ranking over the permitted actions with a *as the label. No reinforcement learning is used.** A supervised policy trained from controlled replays is sufficient for this problem and keeps the learned component narrow and inspectable.

The replay design also answers the usual objection to learned remediation. In live operation, the outcome of an action that was not taken is unobservable, so a policy cannot be trained on what would have happened. Under controlled injection, the same scenario can be replayed with each candidate action, which yields exactly the per-action outcomes the utility needs.

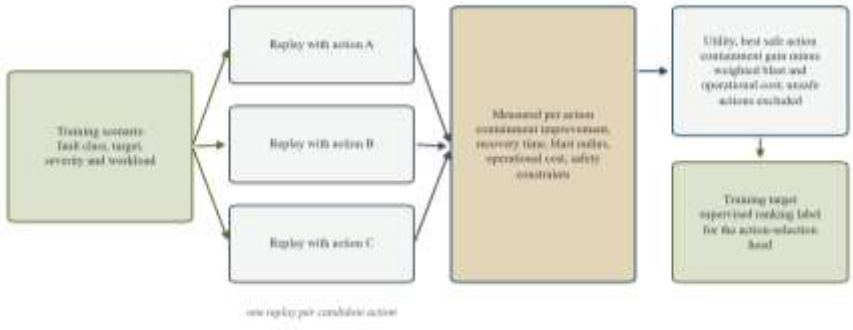


Fig 4: Action-Policy Training. Each Training Scenario Is Replayed with Every Candidate Action, the Outcomes are scored by the Utility, and the Best Safe Action becomes the Supervised Label

The reactive policy used for comparison is the rule engine of the rule-based baseline: a fixed mapping from the dominant anomaly of the focus instance to an action (pool exhaustion to drain and replace, saturation to scale-out, timeouts to load shedding, errors to circuit break, latency to reroute, queueing to deferral), triggered by the predictor instead of by observed alerts. Pairing the same rule engine with different predictors separates better prediction from better action selection (Section 4.4).

3.5. Abstention Mechanism

The controller abstains when its confidence in a predicted focus falls below a threshold τ, following the reject-option principle that a classifier should decline to decide where its error rate would be unacceptable [33]. Confidence is the share of 20 Monte Carlo dropout samples of the breach head in which the focus instance's calibrated probability is at least 0.50. An instance already in observed violation has confidence 1 by construction, because existing reactive alerting would act on it anyway. When the controller abstains, no Tier 2 or Tier 3 action is taken; Tier 1 remains available and the case is escalated. The threshold is selected on validation runs only, as the smallest τ in a grid from 0.50 to 0.95 whose incorrect-action rate is at most 0.10, or failing that the τ with the lowest incorrect-action rate.

3.6. DORA-Aligned Evidence Generation

Every decision writes an immutable record: the timestamp, the graph snapshot, the node features, the predicted breach set and its confidence, the gating rule evaluated, the tier authorized, any hard block applied, the action taken, suppressed or abstained from, and the observed outcome. The reference implementation writes this record for every decision of the full controller on the test runs. It gives a supervisor a machine-generated detection timestamp, a documented and deterministic decision rule rather than an opaque model output, a complete record of suppressed actions that shows the market-session constraints binding, and a reproducible testing record, because the fault-injection harness that trains the model also exercises the platform's resilience.

Table 1 maps the components of the pipeline to the DORA provisions their evidence relates to. The mapping

states alignment, not compliance. Each component supports the corresponding DORA requirement, or produces technical evidence aligned with relevant DORA provisions; whether an entity complies is a determination about the entity, its policies and its governance, which no architecture can make.

Table 1: Pipeline Components and the DORA Provisions Their Evidence Aligns with

Pipeline Component	DORA Alignment
Dependency and graph discovery	Article 8, Identification, in particular 8(4), mapping ICT assets and their interdependencies, and 8(5), mapping interconnections that support critical functions; Article 6, the ICT risk management framework, as the umbrella
Failure and anomaly detection	Article 10, Detection, in particular 10(1) and 10(2); Article 17(3)(a), early warning indicators within the incident management process
Self-healing response	Article 11, Response and recovery, and Article 12, backup policies and restoration and recovery procedures
Chaos and resilience testing	Articles 24 and 25; Article 25(1) names scenario-based tests and end-to-end testing among the required testing types
Threat-led penetration testing	Article 26, advanced testing based on TLPT, and Article 27, requirements for testers; applies only where TLPT is actually involved
Incident classification and reporting	Article 17, incident management process; Article 18, classification; Article 19, reporting; classification criteria in [30]
Third-party dependency tracking	Articles 28 to 30, key principles for managing ICT third-party risk

3.7. Reproducibility Configuration

Table 2 collects the configuration needed to reproduce the controller and its evaluation. All models run on a reverse-mode automatic-differentiation engine written for this study in numpy, whose gradients were checked against central finite differences (maximum relative error below 2×10^{-6}). Every seed derives from one global seed, and a rerun reproduces every number exactly; a check that reran a sample of open-loop and closed-loop runs found identical breach onsets and identical decision logs.

Table 2: Reproducibility Configuration

Parameter	Value
Control interval	1 s
Telemetry window	60 s
Prediction horizon	10 windows (10 min)
Causal view: variable and lags	Composite health score per instance; lags of 1 to 3 windows
Causal view: test and significance	Partial correlation, two-sided Fisher z-test, $\alpha = 0.01$
Causal view: conditioning depth and constraint	At most 2; candidate parents within 2 hops in the dependency view
Causal view: refresh and stability	Every 10 windows from the trailing 30; edge kept if present in 2 consecutive refreshes
GNN layers and attention heads	1 relation-specific attention layer per snapshot; 2 heads
Hidden and GRU dimensions	32 and 32
Dropout	0.2 on the input of the output heads
Loss	Class-weighted focal loss ($\gamma = 2$); blast-radius focal loss and pairwise order loss added
Optimizer and learning rate	Adam, 0.003, gradient-norm clipping at 5
Batch size and epochs	16 runs per step; 12 epochs, best epoch on validation PR-AUC
Train / Validation / Test runs	1,350 / 450 / 450 (225 / 75 / 75 scenarios)
Unseen fault-target combinations in test	3 combinations, 45 scenarios, 270 runs
$\lambda 1$ and $\lambda 2$	0.1 and 1.0
Abstention threshold τ and criterion	0.95; no τ met the 0.10 target, so the lowest validation incorrect-action rate was taken
Nodes	56 instance slots, 30 to 42 serving
Edges per window, mean	178.7 dependency, 139.6 call, 1.3 causal-dependency
Model parameters	20,543 (ST-GNN); 3,335 (action policy)
CPU and RAM	AMD Ryzen 5 7520U, 4 cores and 8 threads; 15.3 GB
GPU	None used

Operating system	Windows 11
Software	Python 3.12.10, numpy 2.5.1, scipy 1.18.1; no PyTorch, no CUDA
Inference batch size	1
Latency measurement	2,400 timed decisions per controller after 300 warm-up decisions

4. Experimental Setup

4.1. Simulated Platform

The evaluation runs on a discrete-time simulation of a generic trading platform, implemented for this paper and released with it. Production experience contributes the problem formulation, meaning the failure taxonomy, the dependency patterns that cascade in trading infrastructure and the operational constraints remediation has to respect during market hours. It contributes no data, and the simulated topology is recognizable as a class of system and traceable to no particular one.

The platform has five functional domains and fourteen services: the order gateway with order routing; market data with three market-data adapters, pricing and reference data; risk with pre-trade risk checks and a risk-limits datastore; matching with the matching engine and order book; and settlement with position keeping, settlement and a trade store. The matching engine, order book and position keeper are stateful services on the live order path. Nineteen typed edges carry traffic, and six more are declared but carry none. Load is modeled per one-second control interval as a fluid queue: arrivals from external load and from callers, capacity limited by processor, thread pool and connection pool, deadlines that turn long waits into errors, latency percentiles that grow with utilization and backlog, timeouts that trigger retries and so amplify the callee's load, and garbage-collection pauses that grow with heap occupancy. An autoscaler adds instances with a warm-up of two to five minutes and removes them when load falls. Rolling restarts occur in a fifth of runs, and every run carries an opening and a closing auction and, in a quarter of runs, a volatility halt.

Fourteen services do not make a fourteen-node graph. The graph is built over service instances, and Table 3 reports its scale.

Table 3: Testbed Scale, Measured Over All 2,250 Runs without a Controller

Quantity	Value
Deployed microservices	14
Runtime service instances, minimum to maximum across workloads	30 to 42
Containers per run, mean	37.4 (one container per instance)
Dependency edges per window, mean	178.7
Call edges per window, mean	139.6
Causal-dependency edges per window, mean	1.3
Graph density, mean (dependency / call / causal)	0.138 / 0.108 / 0.001
Topology changes per run, mean	2.7 without a controller; 4.2 under the full controller

4.2. Workloads and Fault Injection

The fault taxonomy is drawn from production failure modes rather than from what is convenient to inject. It has the five classes in Table 4, each with five severity levels and five eligible target services. Each class is injected under three workload profiles at 35, 55 and 72 percent of nominal capacity, and each combination of class, severity and workload is repeated thirty times with its own seed, target instance and onset time. The design therefore gives $5 \times 5 \times 3 \times 30 = 2{,}250$ runs per method, and that figure applies throughout the paper. Every run lasts sixty one-minute windows after a healthy pre-roll. Severities were calibrated before any model was trained, so that the lowest levels mostly produce near-misses and the highest always breach.

Table 4: Fault Classes, Mechanisms and Severity Ranges

Fault Class	Mechanism	Severity Range, Levels 1 To 5
Resource saturation	Processor hog plus a memory leak at the target instance	15 to 63 percent of capacity lost
Pool exhaustion	Leak of the target's thread and connection pools	55 to 98 percent of the pools leaked
Network faults	Latency injected on calls into the target, or a partial partition	4 to 55 ms added, or 0.2 to 2.5 percent of calls lost
Slow dependency responses	Processing time of the target multiplied	1.6 to 8.5 times
Cascading timeout storms	Target latency raised above its callers' timeouts for a hold period, after which any storm is sustained by retries	15 to 120 s hold

4.3. Training, Validation and Test Splitting

Chaos experiments produce many similar runs, and a random split at the level of the individual run would leak: the same network partition at the same severity, on the same service under the same workload, could appear in both training and test data. Chaos experiments were partitioned into training, validation, and test sets at the scenario level rather than the individual-run level, preventing repeated realizations of the same fault-target-severity-load combination from appearing across training and evaluation sets. A scenario is a combination of fault class, target service, severity and workload, and the six runs of a scenario always stay together. In addition, three combinations of fault class and target service, saturation of the matching engine, a timeout storm at a market-data adapter and pool exhaustion at the position keeper, were drawn before any experiment and withheld from training and validation entirely. They account for 270 of the 450 test runs, so most of the test set consists of faults the models never saw. The split is 1,350 training, 450 validation and 450 test runs.

4.4. Baselines

Predictors: Five baselines are compared with the proposed model, all with the same encoder, hidden size and breach head. An always-negative predictor gives the floor: zero recall and a PR-AUC equal to the base rate. A per-instance recurrent model runs a gated recurrent unit over each instance's own telemetry, with no graph. A topology-free model applies a multilayer perceptron to the current window's features, with no edges and no memory. A temporal graph neural network applies graph convolution over the fixed normalized adjacency of the dependency view [13], followed by a recurrent layer, in the line of [3]. DyGT, a simplified dynamic graph transformer in the line of [10], combines attention over each instance's dependency-view neighbors with attention over its last eight snapshot embeddings, and serves as the representative strong graph-based detection baseline.

Controllers: Six closed-loop configurations are compared on the same runs. No controller is the reference. The rule-based controller maps observed threshold alerts, each sustained for two windows, to fixed actions through the rule engine of Section 3.4. The temporal GNN, DyGT and ST-GNN with reactive policy use that same rule engine, triggered by each model's predicted breaches, so differences among them come from prediction alone. The ST-GNN with learned action policy is the full controller, and its difference from the ST-GNN with reactive policy comes from action selection alone. Every controller passes through the same deterministic gate.

4.5. Metrics and Statistics

Prediction is measured on the test runs without a controller, over 751,543 node-windows of which 2.6 percent are positive, by precision, recall and F1 at the operating threshold chosen on validation data, and by PR-AUC. Recovery is measured on the 247 test runs whose uncontrolled replay produces at least one breach, which are the incident runs every controller faces. MTTR is the simulated time from the first breach to the start of three consecutive windows with no instance in violation, measured at the resolution of one 60 s window; runs that never recover are censored at the end of the run. Blast radius is the number of instances that breached, with a prevented incident counting zero. Containment is the fraction of incident runs in which no instance outside the set already degraded when the controller took its first Tier 2 or Tier 3 action went on to breach. An action is incorrect when its target was not in a breach episode at any point in the following ten windows of the uncontrolled replay. Decision latency is the wall-clock time of each controller's complete decision step at batch size 1. Every result is reported with a 95 percent confidence interval from a percentile bootstrap with 2,000 resamples over test scenarios, so that all windows of a scenario move together, and differences between configurations are reported as paired bootstrap intervals over the same runs.

5. Results

5.1. Prediction Accuracy

Table 5 reports failure-propagation prediction, separately from any recovery outcome. Against a base rate of 0.026, every graph and recurrent model reached a PR-AUC between seven and ten times the base rate. The proposed ST-GNN reached an F1 of 0.228 (95% CI [0.193, 0.260]). Paired over the same test windows, it was ahead of the topology-free model on both F1 (by 0.038, 95% CI [0.019, 0.060]) and PR-AUC (by 0.073, [0.047, 0.097]), ahead of DyGT on PR-AUC (by 0.037, [0.016, 0.058]) and of the temporal GNN on F1 (by 0.027, [0.009, 0.046]). It was not separable from the per-instance recurrent model on either measure, and the temporal GNN had the highest PR-AUC in Table 5, 0.251, with the best recall at a precision of 0.8. No single predictor dominated

Table 5: Failure-Propagation Prediction on the Test Runs (Table A), With 95% Confidence Intervals

Model	Precision	Recall	F1	PR-AUC
Always-negative	n/a	0	0	0.026 [0.019, 0.033] (base rate)
Per-instance GRU	0.334 [0.267, 0.398]	0.170 [0.142, 0.198]	0.225 [0.192, 0.256]	0.235 [0.191, 0.279]
Topology-free MLP	0.243 [0.182, 0.312]	0.155 [0.134, 0.179]	0.189 [0.163, 0.212]	0.155 [0.119, 0.190]
Temporal GNN	0.377 [0.301, 0.454]	0.136 [0.115, 0.158]	0.200 [0.172, 0.226]	0.251 [0.201, 0.299]
DyGT	0.321 [0.265, 0.372]	0.166 [0.141, 0.192]	0.218 [0.190, 0.245]	0.190 [0.148, 0.230]
Proposed ST-GNN	0.315 [0.245, 0.387]	0.178 [0.150, 0.211]	0.228 [0.193, 0.260]	0.228 [0.183, 0.272]

The auxiliary heads carried real signal. The blast-radius head reached a PR-AUC of 0.311 after fault onset against a base rate of 0.023, and the order head ranked pairs of future breachers in the correct order 72.7 percent of the time over 28,620 pairs.

5.2. End-To-End Recovery

Table 6 reports the complete controllers on the 247 incident runs. Three results stand out. First, every predictive controller prevented incidents that the rule-based controller did not: 11.3 to 13.0 percent of incident runs ended with no breach at all, against 1.6 percent for rules acting on observed alerts. Second, under the same reactive rule engine the three predictors produced statistically indistinguishable recovery; the paired difference in blast radius between the ST-GNN and the temporal GNN was −0.03 instances (95% CI [−0.23, 0.17]), and between the ST-GNN and DyGT 0.05 instances ([−0.10, 0.19]). Third, the reactive rule engine itself was the weak link. Its actions on instances already in violation were incorrect 69 percent of the time, and with it every predictor left the mean time to recovery and the blast radius slightly worse than taking no action at all, although the intervals overlap.

Table 6: Complete Self-Healing Controllers on The 247 Incident Test Runs (Table B), With 95% Confidence Intervals. MTTR in Simulated Seconds; Blast Radius in Instances; Latency as the Mean Decision Latency

Controller	MTTR (s)	Blast Radius	Containment	Prevented	Latency
No controller (reference)	817 [658, 976]	7.35 [6.01, 8.70]	0.13 [0.04, 0.22]	0	n/a
Rule-based	930 [755, 1,112]	7.94 [6.47, 9.42]	0.50 [0.39, 0.62]	0.016 [0.004, 0.033]	0.17 ms
Temporal GNN + reactive policy	1,012 [817, 1,209]	8.00 [6.47, 9.44]	0.48 [0.37, 0.59]	0.113 [0.057, 0.179]	0.84 ms
DyGT + reactive policy	981 [802, 1,157]	7.91 [6.39, 9.37]	0.47 [0.36, 0.59]	0.113 [0.057, 0.179]	1.67 ms
ST-GNN + reactive policy	988 [806, 1,172]	7.97 [6.46, 9.42]	0.47 [0.35, 0.58]	0.121 [0.066, 0.188]	1.23 ms
ST-GNN + learned action policy	808 [650, 968]	6.34 [5.02, 7.72]	0.48 [0.34, 0.61]	0.130 [0.057, 0.212]	1.37 ms

The learned action policy changed the outcome: With the predictor held fixed, replacing the reactive rule engine by the learned policy reduced the blast radius by 1.63 instances (paired 95% CI [0.83, 2.50]) and, on the 199 runs that breached under both, the mean time to recovery by 140 s (95% CI 75 to 213 s). It did so mostly by acting less. The policy took 0.28 Tier 2 or Tier 3 actions per run against 1.73 for the reactive policy, because the replays had taught it that most interventions made things worse: in the training replays, the label was the passive Tier 1 action in 1,336 of the 1,626 decision situations, and every action type had a negative mean utility. On held-out training scenarios the policy chose the replay-optimal action 84.6 percent of the time, with a mean regret of 0.119 utility units against 0.188 for always choosing Tier 1. Containment did not change significantly (paired difference 0.016, [−0.098, 0.130]). The full controller was the only configuration whose blast radius fell below that of no controller.

5.3. Ablation Study

Several components change at once between the baselines and the full controller, so Table 7 removes them one at a time. Rows one to four pair each predictor variant with the same reactive rule engine; the last two rows differ only in action selection.

Table 7: Ablation on the Test Runs. Propagation F1 With 95% CI, MTTR in Simulated Seconds, and Containment Ratio

Configuration	Propagation F1	MTTR (s)	Containment
Dependency graph only	0.227 [0.198, 0.256]	1,002 [817, 1,191]	0.48 [0.37, 0.59]
Dependency + call graph	0.249 [0.212, 0.284]	1,033 [841, 1,224]	0.45 [0.34, 0.57]
Dependency + call + causal	0.228 [0.193, 0.260]	988 [806, 1,172]	0.47 [0.35, 0.58]
Full graph without GRU	0.200 [0.171, 0.226]	1,014 [822, 1,206]	0.49 [0.38, 0.60]
ST-GNN + reactive action policy	0.228 [0.193, 0.260]	988 [806, 1,172]	0.47 [0.35, 0.58]
ST-GNN + learned action policy	0.228 [0.193, 0.260]	808 [650, 968]	0.48 [0.34, 0.61]

The ablation answers the three questions it was designed for. Temporal modeling helps. Removing the recurrent layer reduced F1 by 0.028 (95% CI [0.018, 0.038]) and PR-AUC by 0.058 ([0.036, 0.078]), the largest single effect on prediction. The call view helps; the causal-dependency view, as built here, does not. Adding the call view to the dependency view raised F1 by 0.022 ([0.001, 0.044]), and the dependency-plus-call configuration was the strongest predictor in the study, at 0.249. Adding the causal-dependency view then lowered F1 by 0.022 ([0.004, 0.041]). That view averaged 1.3 edges per window, because in the simulation degradation crosses between instances within a

single 60 s window, and a search over lags of one window or more finds few dependencies that survive conditioning on an instance's own past. The action policy improves recovery independently of prediction, as Section 5.2 showed; no change to the predictor moved MTTR or blast radius by a comparable amount.

5.4. Decision Latency

Table 8 reports the wall-clock decision latency of each controller on the hardware in Table 2, at batch size 1, over 2,400 timed decisions after 300 warm-up decisions were excluded. Each timed step covers feature normalization, snapshot construction across all three views, one model step, calibration, the gate, action selection and, for the full controller, the twenty-sample abstention check. The periodic refresh of the causal-dependency view, every ten windows, took 4.5 ms on average and is reported separately.

Table 8: Decision Latency per Controller, in Milliseconds

Controller	Mean [95% CI]	p50	p95
Rule-based	0.17 [0.16, 0.18]	0.15	0.27
Temporal GNN + reactive policy	0.84 [0.79, 0.89]	0.74	1.54
DyGT + reactive policy	1.67 [1.59, 1.75]	1.44	2.90
ST-GNN + reactive policy	1.23 [1.19, 1.26]	1.10	2.02
ST-GNN + learned action policy	1.37 [1.30, 1.44]	1.23	2.27

The controller operates on the resilience control plane rather than the latency-critical order-execution path. Its 1.37 ms mean decision latency remains within the testbed's automated-remediation budget, which is one 60 s control window. These figures describe a numpy implementation on a laptop processor without a GPU. No LLM-based controller was evaluated, so no latency comparison with LLM-based remediation is made, and the latency of such a controller would depend on model size, hardware, quantization, token count and deployment architecture.

5.5. Abstention and Distribution Shift

Distribution shift was tested on the 270 runs of withheld fault-target combinations, executed on a perturbed topology: two additional traffic dependencies, an extra market-data instance, and per-run variation of capacity, base latency, fanout and timeouts. On the withheld combinations at nominal topology, the ST-GNN's F1 fell from 0.265 on seen combinations to 0.205, a measured degradation of 0.060, and it’s PR-AUC from 0.253 to 0.220. Under the perturbed topology, the full controller's containment fell from 0.54 to 0.40 and its blast radius rose from 7.4 to 10.6 instances, against 11.0 with no controller.

Table 9: Abstention Threshold Sensitivity under Distribution Shift, Full Controller

Threshold	Controller Coverage	Successful Containment	Incorrect Actions
None (abstention off)	1.00	0.40 [0.27, 0.54]	0.63 [0.45, 0.81]
0.50	1.00	0.40 [0.27, 0.54]	0.63 [0.45, 0.81]
0.60	1.00	0.40 [0.27, 0.54]	0.63 [0.45, 0.81]
0.70	1.00	0.40 [0.27, 0.54]	0.63 [0.45, 0.81]
0.80	1.00	0.40 [0.27, 0.54]	0.63 [0.45, 0.81]
0.95 (selected)	1.00	0.40 [0.27, 0.54]	0.63 [0.45, 0.81]

Abstention never triggered under shift: The abstention rate at the selected threshold was zero, and Table 9 is identical at every threshold. The reason is structural rather than statistical. Of the 128 Tier 2 and Tier 3 actions the full controller applied on the test runs, 127 targeted an instance already in observed violation, where confidence is 1 by construction, and only one acted on a prediction alone. Because the learned policy had already learned to act rarely on predictions, abstention had no decisions left to decline. On the validation runs, where it did engage, coverage fell to 0.82 at $\tau = 0.95$ without a meaningful change in the incorrect-action rate (0.107 against 0.108 with abstention off), so no threshold in the grid met the 0.10 target. Performance with abstention off is the first row of Table 9 and is the same.

6. Discussion

6.1. Why Graph Reasoning Helps, and Where It Stops Helping

The graph models' advantage over the topology-free baseline was large and clear, and it came from two sources the ablation separates: memory, which the recurrent layer supplies, and structure, which the call view supplies. A node whose own metrics look clean can be the next to fail because its upstream neighbor is degrading, and a model given the same features without edges or history cannot see it. The advantage over a per-instance recurrent model was not established, which says that, in this simulation, much of the predictive signal is already present in an instance's own trajectory of leading indicators. The richer views did not all pay their way. The call view, which records what actually happened in the window, helped; the causal-dependency view, which tries to infer influence from lagged statistics, added little at a 60 s window because the dynamics it would

need to detect play out faster than its smallest lag. A shorter window or a contemporaneous search step would be the natural next design, and the result is reported as found rather than tuned away.

6.2. Prediction versus Action

The clearest finding is that a better prediction does not by itself produce a better recovery. Three predictors with different accuracy produced the same recovery when they drove the same rule engine, and that rule engine, applied to instances already in violation, chose harmful actions more often than helpful ones. The learned policy changed the outcome because it was trained on what each action actually did in replays of the same scenario, and what it learned was restraint: on most decisions the best safe action was to pre-warm capacity and escalate. That is the cost asymmetry of Section 3.4 made measurable. A missed intervention returns the platform to the status quo, while a wrong intervention on a degraded platform can widen the incident, and a controller that has measured that asymmetry acts less and recovers faster.

6.3. What Abstention Can and Cannot Do

Abstention guards decisions made on predictions. In this controller, almost every intervention was made on an instance already in violation, where the existing alerting would act anyway and confidence is 1 by design, so abstention had nothing to decline. Two readings follow. The design is sound for the decisions it covers, and its effect will appear only in a controller that intervenes on predictions more often, for example with a lower trigger threshold or a policy trained to act earlier. And the reject option should extend to observed-violation decisions as well, using the policy's own uncertainty about which action helps, since that is where this controller's incorrect actions occurred.

6.4. Predictive Capacity Actions

Pre-warming in Tier 1 and scale-out in Tier 3 are capacity actions taken on a prediction. Their value depends on lead time: capacity that becomes ready after the breach is reactive autoscaling under another name. The policy's preference for pre-warming, and the predictive controllers' prevention of incidents that rules acting on alerts did not prevent, are both consequences of acting within the ten-minute horizon. This is the point at which resource management and failure management meet on this class of platform, and the boundary between this paper and the author's earlier work on SLA-aware allocation [8].

6.5. DORA-Aligned Evidence

The evidence record makes three things inspectable that are otherwise reconstructed after the fact. The detection timestamp is recorded by the system rather than asserted by a person. The decision rule that authorized an action is published and deterministic. And the record of suppressed and abstained actions shows the market-session constraints and the safety gate binding in practice, not only in policy. The architecture produces an evidence artifact relevant to the corresponding requirements of Table 1. It produces technical evidence supporting operational resilience testing through the chaos harness, which here doubles as the training and evaluation harness. Whether those artifacts are sufficient for a particular entity remains a matter for that entity's governance and its supervisor.

7. Limitations

- Simulation, not deployment: Every result comes from a discrete-time simulation of a generic trading platform. Queueing is modeled as a fluid approximation at one-second resolution, latency percentiles are model-based, the call view is computed from simulated call volumes rather than sampled from distributed traces, and cross-service effects lag by one control interval. The simulation is not a deployment of a public microservice benchmark and not a pre-production environment, and its findings are claims about the architecture under these dynamics.
- Synthetic chaos-generated labels: Every label comes from injected faults. Injection gives exact onset and known attribution, but injected faults are cleaner than real ones, and a real cascading failure may combine causes the taxonomy does not contain.
- Limited real cascading failures: The model has not been evaluated against production failures, which are rare and whose records are not publishable.
- Distribution shift and cross-environment generalization: Performance degraded on withheld fault-target combinations and further on a perturbed topology (Section 5.5). A different platform, workload mix or tenant brings shifts the experiment did not test.
- The causal-dependency view: As implemented, with one composite variable per instance, lags of one to three windows and the first stage of PCMCI only, the view was very sparse and did not improve prediction. The finding is specific to that construction and window length. The view is in any case a statistical construct, and its edges do not establish causal mechanisms.
- Simplified baselines and model: DyGT is a simplified dynamic graph transformer without the neighbor co-occurrence encoding and patching of DyGFormer. The ST-GNN uses one attention layer per snapshot and fuses relations by summation.
- The learned policy: Its labels are the best single action at the first two decisions of each training run, so sequences of actions are not optimized, and its decision situations were drawn from the predictor's own training runs, where predictions may be more accurate than on new data. The policy chooses the action, while the focus instance is chosen by a fixed rule.
- Abstention: In this evaluation abstention never engaged under shift, for the structural reason given in Section 5.5, and its benefit is therefore not demonstrated.
- No LLM baseline and no live deployment: No LLM-based controller was evaluated. The controller

has not acted on a live trading platform, operator escalations have no effect in the simulation, and the market-session constraints have not been exercised against real auctions, volatility halts or operator responses.

8. Conclusion

This paper integrated multi-view spatiotemporal failure-propagation modeling, bounded cost-sensitive remediation and DORA-aligned evidence generation into a closed-loop resilience controller for financial trading microservices, and evaluated it in a reproducible simulation of a fourteen-service trading platform with chaos-injected faults, at 2,250 runs per method under a scenario-level split. Graph and temporal modeling predicted breaches far above the base rate, with the recurrent layer and the call view contributing measurably and the causal-dependency view, as built, contributing nothing. Prediction alone did not improve recovery. The component that did was the learned action policy, trained from counterfactual replays: with predictions held fixed, it reduced the blast radius by 1.63 instances and the mean time to recovery by 140 s, chiefly by declining interventions that the replays showed to be harmful, at a mean decision latency of 1.37 ms. The evidence record supports the corresponding DORA requirements by making detection, decision and suppression inspectable; it does not establish compliance. Evaluation on benchmark deployments, against real cascading failures and in live operation is the work that remains.

References

1. European Parliament and Council of the European Union, "Regulation (EU) 2022/2554 of the European Parliament and of the Council of 14 December 2022 on digital operational resilience for the financial sector," Official Journal of the European Union, L 333, pp. 1-79, Dec. 27, 2022. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2022/2554/oj (accessed Oct. 15, 2026).
2. B. Yu, H. Yin, and Z. Zhu, "Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting," in Proc. 27th Int. Joint Conf. Artificial Intelligence (IJCAI), 2018, pp. 3634-3640, doi: 10.24963/ijcai.2018/505.
3. L. Zhao, Y. Song, C. Zhang, Y. Liu, P. Wang, T. Lin, M. Deng, and H. Li, "T-GCN: A temporal graph convolutional network for traffic prediction," IEEE Transactions on Intelligent Transportation Systems, vol. 21, no. 9, pp. 3848-3858, 2020, doi: 10.1109/TITS.2019.2935152.
4. L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in Proc. IEEE/IFIP Network Operations and Management Symp. (NOMS), 2020, pp. 1-9, doi: 10.1109/NOMS47738.2020.9110353.
5. J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," ACM Computing Surveys, vol. 55, no. 3, pp. 1-39, 2022, doi: 10.1145/3501297.
6. J. Kephart and D. Chess, "The vision of autonomic computing," Computer, vol. 36, no. 1, pp. 41-50, 2003, doi: 10.1109/MC.2003.1160055.
7. H. Psaier and S. Dustdar, "A survey on self-healing systems: Approaches and systems," Computing, vol. 91, no. 1, pp. 43-73, 2011, doi: 10.1007/s00607-010-0107-y.
8. S. Uchdadiya, "Deep reinforcement learning for SLA-aware cloud resource management in financial trading platforms," International Journal of Artificial Intelligence and Machine Learning, vol. 6, no. 12s, pp. 332-350, 2026. [Online]. Available: https://www.svedbergopen.com/index.php/ijaiml/article/view/2422 (accessed Oct. 15, 2026).
9. A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. Schardl, and C. Leiserson, "EvolveGCN: Evolving graph convolutional networks for dynamic graphs," in Proc. AAAI Conf. Artificial Intelligence, vol. 34, no. 4, 2020, pp. 5363-5370, doi: 10.1609/aaai.v34i04.5984.
10. L. Yu, L. Sun, B. Du, and W. Lv, "Towards better dynamic graph learning: New architecture and unified library," in Advances in Neural Information Processing Systems, vol. 36, 2023, doi: 10.52202/075280-2960.
11. P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in Proc. Int. Conf. Learning Representations (ICLR), 2018, arXiv:1710.10903.
12. M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in The Semantic Web (ESWC 2018), Lecture Notes in Computer Science, Springer, 2018, pp. 593-607, doi: 10.1007/978-3-319-93417-4_38.
13. T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in Proc. Int. Conf. Learning Representations (ICLR), 2017, arXiv:1609.02907.
14. K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in Proc. 2014 Conf. Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1724-1734, doi: 10.3115/v1/D14-1179.
15. Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in Proc. 24th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019, pp. 19-33, doi: 10.1145/3297858.3304004.
16. C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An end-to-end troubleshooting framework for microservices on multi-source data," in Proc. IEEE/ACM 45th Int. Conf. Software Engineering (ICSE), 2023, pp. 1750-1762, doi: 10.1109/ICSE48619.2023.00150.
17. Z. Li, N. Zhao, M. Li, X. Lu, L. Wang, D. Chang, X. Nie, L. Cao, W. Zhang, K. Sui, Y. Wang, X. Du, G. Duan, and D. Pei, "Actionable and interpretable fault

localization for recurring failures in online service systems," in Proc. 30th ACM Joint European Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE), 2022, pp. 996-1008, doi: 10.1145/3540250.3549092.

18. Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ML-driven performance debugging in microservices," in Proc. 26th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021, pp. 135-151, doi: 10.1145/3445814.3446700.
19. X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," IEEE Transactions on Software Engineering, vol. 47, no. 2, pp. 243-260, 2021, doi: 10.1109/TSE.2018.2887384.
20. T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "Recommending root-cause and mitigation steps for cloud incidents using large language models," in Proc. IEEE/ACM 45th Int. Conf. Software Engineering (ICSE), 2023, pp. 1737-1749, doi: 10.1109/ICSE48619.2023.00149.
21. Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, J. Zeng, S. Ghosh, X. Zhang, C. Zhang, Q. Lin, S. Rajmohan, D. Zhang, and T. Xu, "Automatic root cause analysis via large language models for cloud incidents," in Proc. 19th European Conf. Computer Systems (EuroSys), 2024, pp. 674-688, doi: 10.1145/3627703.3629553.
22. P. Notaro, J. Cardoso, and M. Gerndt, "A survey of AIOps methods for failure management," ACM Transactions on Intelligent Systems and Technology, vol. 12, no. 6, pp. 1-45, 2021, doi: 10.1145/3483424.
23. X. Ma, J. Wu, S. Xue, J. Yang, C. Zhou, Q. Z. Sheng, H. Xiong, and L. Akoglu, "A comprehensive survey on graph anomaly detection with deep learning," IEEE Transactions on Knowledge and Data Engineering, vol. 35, no. 12, pp. 12012-12038, 2023, doi: 10.1109/TKDE.2021.3118815.
24. P. Spirtes and C. Glymour, "An algorithm for fast recovery of sparse causal graphs," Social Science Computer Review, vol. 9, no. 1, pp. 62-72, 1991, doi: 10.1177/089443939100900106.
25. J. Runge, P. Nowack, M. Kretschmer, S. Flaxman, and D. Sejdinovic, "Detecting and quantifying causal associations in large nonlinear time series datasets," Science Advances, vol. 5, no. 11, Art. no. eaau4996, 2019, doi: 10.1126/sciadv.aau4996.
26. P. Chen, Y. Qi, P. Zheng, and D. Hou, "CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in Proc. IEEE INFOCOM, 2014, pp. 1887-1895, doi: 10.1109/INFOCOM.2014.6848128.
27. 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, 2016, doi: 10.1109/MS.2016.60.
28. A. Basiri, L. Hochstein, N. Jones, and H. Tucker, "Automating chaos experiments in production," in Proc. IEEE/ACM 41st Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 31-40, doi: 10.1109/ICSE-SEIP.2019.00012.
29. Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in Proc. 24th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019, pp. 3-18, doi: 10.1145/3297858.3304013.
30. European Commission, "Commission Delegated Regulation (EU) 2024/1772 supplementing Regulation (EU) 2022/2554 with regard to regulatory technical standards specifying the criteria for the classification of ICT-related incidents and cyber threats, setting out materiality thresholds and specifying the details of reports of major incidents," Official Journal of the European Union, 2024. [Online]. Available: https://eur-lex.europa.eu/eli/reg_del/2024/1772/oj (accessed Oct. 15, 2026).
31. T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 2, pp. 318-327, 2020, doi: 10.1109/TPAMI.2018.2858826.
32. T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets," PLOS ONE, vol. 10, no. 3, Art. no. e0118432, 2015, doi: 10.1371/journal.pone.0118432.
33. C. Chow, "On optimum recognition error and reject tradeoff," IEEE Transactions on Information Theory, vol. 16, no. 1, pp. 41-46, 1970, doi: 10.1109/TIT.1970.1054406.