International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I2P108
Eureka Vision Publication | Volume 7, Issue 2, 49-57, 2026

Original Article

Digital Twin Simulation of Enterprise Healthcare Integration Environments: A Virtual Modeling Framework for Pre-Deployment Validation, Capacity Planning, and Failure Scenario Rehearsal

Sindhukumar Sundaram
Tennessee, USA.

Received On: 20/02/2026 Revised On: 25/03/2026 Accepted On: 02/04/2026 Published On: 10/04/2026

Abstract - Enterprise healthcare integration environments are complex distributed systems comprising hundreds of interdependent channels, diverse transport protocols, heterogeneous message standards, and variable upstream/downstream system behaviors. Changes to these environments including new interface deployments, configuration modifications, volume increases, and infrastructure migrations carry inherent risk, as interaction effects between channels sharing common resources are difficult to predict analytically. Currently, no validated methodology exists for simulating the behavior of an entire integration environment prior to production changes. This paper introduces the Integration Environment Digital Twin (IEDT) framework a discrete-event simulation model that replicates the complete topology, resource allocation, message routing logic, and volumetric behavior of an enterprise healthcare integration engine. IEDT enables operators to execute what-if scenarios including new interface onboarding, volume surge simulations, infrastructure failure injections, and configuration change impact assessments in a sandboxed virtual environment before applying changes to production. Validation across three operationally representative scenarios demonstrates that IEDT predicts system-wide message throughput within 6.3% of observed values, identifies resource contention bottlenecks with 89% accuracy, and correctly models cascading queue backpressure across 11 of 12 affected channels during outage simulation. Thread pool exhaustion timing is predicted within 4 minutes of actual occurrence. IEDT provides a validated, low-risk methodology for pre-deployment impact analysis in enterprise healthcare integration environments.

Keywords - Digital Twin, Simulation Modeling, Discrete-Event Simulation, Healthcare Integration, Capacity Planning, Failure Rehearsal, Enterprise Middleware, Pre-Deployment Validation, HL7, FHIR.

1. Introduction

Healthcare integration engines are the operational backbone of clinical data exchange, routing millions of Health Level Seven (HL7) v2.x and Fast Healthcare Interoperability Resources (FHIR) messages daily between electronic health record (EHR) systems, laboratory information systems, pharmacy platforms, radiology systems, registries, and health information exchanges (HIEs) [1][2]. Enterprise-scale health systems routinely operate integration environments comprising hundreds of concurrently active channels distributed across multiple engine instances, data centers, and client facilities.

These environments are not static. New interfaces are deployed regularly as client facilities onboard new systems, regulatory requirements expand data exchange mandates, and organizational mergers consolidate previously independent integration topologies. Each change introduces risk: a new high-volume interface may consume thread pool capacity that adjacent channels depend on; a configuration modification may alter routing behavior in unexpected ways; an infrastructure migration may expose latency characteristics that trigger timeout cascades.

Currently, assessing the impact of changes relies on a combination of institutional knowledge, manual capacity estimation, and staged rollout with reactive monitoring. When estimates are wrong, the consequences manifest as production failures: channel stalls, queue saturation, memory exhaustion, and cascading backpressure that disrupts clinical workflows [3]. The Trusted Exchange Framework and Common Agreement (TEFCA) has driven exchange volumes from roughly 10 million records in January 2025 to nearly 500 million [4], amplifying both the complexity and the risk associated with integration environment changes.

Digital twin technology the creation of virtual replicas of physical systems for simulation, analysis, and optimization has demonstrated significant value in manufacturing [5], aerospace [6], and smart city infrastructure [7]. However, the concept has not been formally applied to healthcare integration middleware. This paper introduces the Integration Environment Digital

Twin (IEDT) framework, addressing three research questions:

- RQ1: Can a discrete-event simulation model accurately replicate the throughput, latency, and resource consumption behavior of an enterprise healthcare integration engine?
- RQ2: Can IEDT predict the impact of new interface deployments, volume surges, and infrastructure failures with sufficient accuracy for operational decision-making?
- RQ3: What modeling fidelity is required to capture inter-channel resource contention and cascading failure dynamics?

Contributions: (1)A formal discrete-event simulation model for healthcare integration engines capturing channel-level processing, shared resource contention, and message routing dynamics; (2)A resource contention model for thread pools, memory heaps, database connections, and network sockets shared across integration channels; (3) Three validated simulation scenarios: planned go-live surge, unplanned system outage, and infrastructure migration; (4) Empirical evaluation demonstrating prediction accuracy sufficient for operational pre-deployment analysis.

2. Background and Related Work

2.1. Digital Twin Concepts and Applications

The digital twin concept originated in manufacturing, defined as a virtual replica of a physical system that mirrors its state, behavior, and lifecycle through continuous data synchronization [5]. Digital twins have been applied to predictive maintenance in industrial equipment [6], network topology optimization in telecommunications [8], and smart building energy management [7].

A digital twin for an integration engine differs from these applications in several important ways. Integration environments exhibit discrete event-driven behavior rather than continuous dynamics; resource contention effects create non-linear performance degradation; and the clinical consequence of simulation inaccuracy demands conservative safety margins.

2.2. Healthcare Integration Engine Architecture

Healthcare integration engines operate on channel-based architectures where each channel defines a complete message processing pipeline [1][2]. Channels share computational resources thread pools, heap memory, database connection pools, and network sockets creating interdependencies. Message types span the full HL7 v2.x suite including Admit, Discharge, Transfer (ADT), Order Entry (ORM), Observation Result (ORU), Medical Document Management (MDM), Detail Financial Transaction (DFT), Billing Account Record (BAR), Master File Notification (MFN), Pharmacy/Treatment Administration (RAS), Pharmacy/Treatment Encoded Order (RDE), and Scheduling Information Unsolicited (SIU) [9], each with distinct volume profiles, processing complexity, and clinical urgency characteristics.

When external service endpoints degrade or become unavailable, reactive mechanisms such as fault-tolerant timeout frameworks prevent indefinite thread blocking and connection holding [10]. However, the *interaction effects* between multiple channels competing for shared resources during degraded conditions are difficult to predict without simulation a single slow destination can propagate thread starvation across dozens of otherwise healthy channels.

2.3. Discrete-Event Simulation in IT Systems

Discrete-event simulation (DES) models systems as sequences of events occurring at discrete time points, advancing simulation time from event to event [11]. DES has been applied to network performance modeling [8], server capacity planning [12], and database workload analysis. Queuing theory particularly M/G/1 and M/G/c models provides the mathematical foundation for modeling message processing systems with variable service times and finite server capacity [13].

2.4. Gap Analysis

No published work applies digital twin or formal simulation methodology to healthcare integration engines. Existing capacity planning approaches for middleware rely on static calculations or vendor-provided sizing guides that do not account for inter-channel resource contention, message-type-specific processing variability, or cascading failure dynamics. IEDT addresses this gap.

3. Methodology

3.1. Research Design

This study follows a design science research (DSR) methodology [14]: problem identification through operational analysis; artifact design (IEDT framework); demonstration across three simulation scenarios; and quantitative validation against observed system behavior. The evaluation measures prediction accuracy for throughput, latency, resource utilization, and failure propagation timing.

3.2. Reference Environment

IEDT is validated against a reference integration environment, summarized in Table I. All telemetry and message data is synthetically generated no protected health information (PHI) is used.

Table 1. Reference Environment Parameters

Parameter	Value
Engine instances	4 (active-active, 2 data centers)
Active channels	160
Client facilities	12
Message types	ADT, ORM, ORU, MDM, DFT, BAR, MFN, RAS, RDE, SIU

Sustained volume	400 msg/sec (aggregate)
Peak volume	1,100 msg/sec
Thread pool (per engine)	200 threads
Heap memory (per engine)	16 GB allocated
DB connection pool (per engine)	100 connections
Hardware (per engine)	8-core Xeon @ 2.6 GHz, 32 GB RAM, NVMe SSD
Data generation	Synthea v3.3.0 [15] message corpus

3.3. IEDT Simulation Model

The IEDT simulation model is built on four interconnected sub-models, as illustrated in Fig. 1.

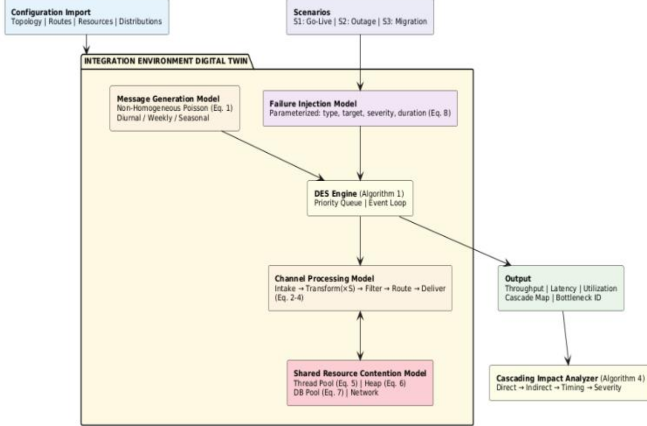

Figure 1. IEDT Architecture

C.1. Message Generation Model

The message generation model produces synthetic message arrivals that match empirical healthcare volume patterns. Arrivals follow a non-homogeneous Poisson process with a time-varying rate function:

$$\lambda(t)=\lambda_{base}\cdot f_{diurnal}(t)\cdot f_{weekly}(t)\cdot f_{seasonal}(t)+\lambda_{burst}(t) \quad (1)$$

Where λ_base is the average arrival rate, f_diurnal captures the characteristic 8 AM–6 PM peak pattern, f_weekly models weekday/weekend differentials, f_seasonal represents census fluctuations, and λ_burst models event-driven surges (e.g., go-live events). Equation (1) enables the simulation to reproduce realistic volume patterns including peak-hour surges and holiday troughs.

Each generated message carries attributes: message type, source system, message size (bytes), segment count, and clinical priority.

C.2. Channel Processing Model

Each channel is modeled as a multi-stage queuing system:

$$T_{channel}(m) = T_{intake}(m) + \sum_{i=1}^{S} T_{transform_i}(m) + T_{filter}(m) + T_{route}(m) + T_{deliver}(m) \quad (2)$$

Where T_channel(m) is the total processing time for message m, and S is the number of transformation stages. Equation (2) decomposes channel latency into its constituent processing phases. Each stage draws its processing time from an empirically fitted distribution:

$$T_{stage}(m) \sim \mathrm{LogNormal}(\mu_{stage,\,type}, \sigma_{stage,\,type}) \quad (3)$$

Where μ and σ are parameterized by message type and stage, fitted from observed processing time distributions. Equation (3) captures the right-skewed latency distributions characteristic of real integration engine processing, where most messages process quickly but occasional complex messages exhibit significantly longer processing times.

Destination delivery time is modeled as:

$$T_{deliver}(m)=T_{network}+T_{destprocessing}+T_{ackwait} \quad (4)$$

Where each component is drawn from distribution profiles specific to the destination system type (EHR, registry, HIE, etc.). Equation (4) separates controllable factors (network) from external factors (destination processing) for more accurate failure injection modeling.

C.3 Shared Resource Contention Model

The resource contention model is the core differentiator of IEDT it captures the non-linear performance degradation that occurs when channels compete for shared resources. Four resource pools are modeled:

Thread Pool: Modeled as a bounded semaphore with FIFO waiting queue. When a channel requires a thread for message processing and the pool is exhausted, the message waits in the thread acquisition queue:

$$W_{thread}(t) = \{\ 0,\ \text{if } a_{active}(t) < P_{size}\ \ (a_{active}(t) - P_{size} + 1) / \mu_{release}(t),\ \text{otherwise}\ \} \quad (5)$$

Where μ_release is the mean thread release rate at time t. Equation (5) models the queuing delay that grows as thread utilization approaches saturation.

Heap Memory: Modeled as a cumulative allocation tracker with garbage collection (GC) events:

$$\mathrm{heap}(t) = \mathrm{heap}(t-1) + \sum_{m\in\mathrm{active}} \mathrm{size}\,(m) - \mathrm{gc_freed}(t) \quad (6)$$

GC events are triggered when heap utilization exceeds configurable thresholds (70%, 85%, 92%), with GC pause times drawn from empirical distributions. Equation (6) captures the sawtooth memory pattern characteristic of Java-based integration engines.

Database Connection Pool: Modeled similarly to the thread pool as a bounded resource with acquisition queuing and configurable timeout:

$$T_{db_wait}(t)=f(pool_util(t),query_duration_dist) \quad (7)$$

Network Sockets: Modeled with configurable bandwidth limits, connection establishment latency, and TCP backpressure effects.

C.4 Failure Injection Model

IEDT supports parameterized failure injection for simulation scenarios:

$$F(t) = \text{failure}(t, \text{type}, \text{target}, \text{severity}, \text{duration}) \quad (8)$$

Where type ∈ {DEST_TIMEOUT, DEST_SLOW, NETWORK_PARTITION, MEMORY_LEAK, ERROR_STORM}, target identifies the affected component (specific channel, destination, or resource pool), severity scales the impact (0.0–1.0), and duration defines the failure period. Equation (8) provides a unified parameterization enabling reproducible failure scenario definition.

3.4. Model Calibration

IEDT model parameters are calibrated through a three-phase process:

- Phase 1 Static Configuration: Channel topology, routing rules, transformation stage counts, and resource pool sizes are imported directly from the reference environment's configuration.
- Phase 2 Distribution Fitting: Processing time distributions for each stage and message type are fitted to empirical telemetry using maximum likelihood estimation (MLE) with Kolmogorov-Smirnov (K-S) goodness-of-fit testing. Table II summarizes the fitted distributions.
- **Phase 3 Behavioral Validation:** The calibrated model is run under baseline conditions (normal production load) and compared against observed telemetry for throughput, latency percentiles, and resource utilization. Parameters are iteratively adjusted until baseline predictions fall within 5% of observed values.

Table 2. Fitted Processing Time Distributions by Stage and Message Type

Stage	Message Type	Distribution	Parameters	K–S *p*
Intake	All	LogNormal	$\mu = 0.8, \sigma = 0.3$	0.87
Transform (map)	ADT	LogNormal	$\mu = 1.2, \sigma = 0.4$	0.82
Transform (map)	ORU	LogNormal	$\mu = 1.8, \sigma = 0.6$	0.79
Transform (map)	MDM	LogNormal	$\mu = 2.4, \sigma = 0.9$	0.74
Transform (vocab)	All	LogNormal	$\mu = 0.6, \sigma = 0.2$	0.91
Filter	All	Exponential	$\lambda = 2.1$	0.88
Route	All	Deterministic	t = 0.5 ms	N/A
Deliver (EHR)	All	LogNormal	$\mu = 3.2, \sigma = 1.1$	0.71
Deliver (Registry)	All	LogNormal	$\mu = 4.8, \sigma = 1.8$	0.68
GC pause (minor)	N/A	LogNormal	$\mu = 1.5, \sigma = 0.5$	0.83
GC pause (major)	N/A	Gamma	$\alpha = 2, \beta = 25$	0.76

3.5. Simulation Scenarios

Three operationally representative scenarios are evaluated, summarized in Table III.

Table 3. Simulation Scenarios

#	Scenario	Description
S1	Planned Go-Live Surge	40 new interfaces added simultaneously, increasing volume by 65%. Simulates EHR go-live at a new client facility.
S2	Unplanned Downstream Outage	Major destination system (serving 12 channels) becomes unavailable for 30 min. Simulates EHR downtime.
S3	Infrastructure Migration	Migration from VM-based deployment to a container-orchestrated environment with different resource allocation profiles.

3.6. Evaluation Metrics

Table IV defines the evaluation metrics. Prediction error is computed as the absolute percentage difference between IEDT-simulated values and reference environment observed values.

Table 4. Evaluation Metrics

Metric	Definition	Target
Throughput prediction error	$\lvert sim - obs \rvert / obs \times 100\%$	< 10%
Latency prediction error (p95)	$\lvert sim_p95 - obs_p95 \rvert / obs_p95$	< 15%
Resource utilization error	$\lvert sim_util - obs_util \rvert$ (absolute percentage points)	< 8 pp
Bottleneck identification accuracy	Correctly identified / total	> 85%
Failure propagation accuracy	Correctly predicted affected channels / actual affected	> 80%
Failure timing error	$\lvert sim_time - obs_time \rvert$	< 5 min

4. Algorithms and Formal Description

4.1. Algorithm 1: IEDT Simulation Engine

Input: config Environment topology and parameters
scenario Failure/change injection definition
duration Simulation time window
Output: metrics Predicted throughput, latency, utilization

```
1:  FUNCTION RunSimulation(config, scenario, duration)
2:      env ← InitializeEnvironment(config)
3:      event_queue ← PriorityQueue()
4:      clock ← 0
5:      // Seed initial message arrivals (Eq. 1)
6:      FOR EACH source IN env.sources DO
7:          t_next ← SampleNHPP(source.λ, clock)
8:          event_queue.push(MessageArrival(source, t_next))
9:      END FOR
10:     // Inject scenario events
11:     FOR EACH injection IN scenario.injections DO
12:         event_queue.push(injection)
13:     END FOR
14:     // Main simulation loop
15:     WHILE clock < duration DO
16:         event ← event_queue.pop()
17:         clock ← event.timestamp
18:         SWITCH event.type:
19:             CASE MessageArrival:
20:                 ProcessArrival(env, event, event_queue)
21:             CASE StageComplete:
22:                 ProcessStageComplete(env, event, event_queue)
23:             CASE ResourceRelease:
24:                 ProcessResourceRelease(env, event, event_queue)
25:             CASE FailureInjection:
26:                 ApplyFailure(env, event)
27:             CASE GarbageCollection:
28:                 ProcessGC(env, event, event_queue)
29:         END SWITCH
30:         CollectMetrics(env, clock)
31:     END WHILE
32:     RETURN AggregateMetrics(env)
33: END FUNCTION
```

Complexity: $O(E \times \log E)$ where E = total events processed, dominated by priority queue operations. For the reference environment, E ≈ 50M events per simulated hour.

4.2. Algorithm 2: Message Arrival Processing

Input: env, event, event_queue
Output: Updated environment state, new events queued

```
1:  FUNCTION ProcessArrival(env, event, event_queue)
2:      msg ← GenerateMessage(event.source)
3:      channel ← RouteToChannel(env, msg)
4:      // Attempt thread acquisition (Eq. 5)
5:      thread ← env.thread_pool.tryAcquire()
6:      IF thread IS NULL THEN
7:          channel.wait_queue.enqueue(msg)
8:          RecordWaitStart(msg, env.clock)
9:      ELSE
10:         // Begin intake stage processing (Eq. 2, 3)
11:         t_intake ← SampleLogNormal(
                μ_intake, σ_intake)
12:         event_queue.push(StageComplete(
                msg, channel, INTAKE, env.clock + t_intake))
13:         // Update heap allocation (Eq. 6)
14:         env.heap.allocate(msg.size)
15:         CheckGCThresholds(env, event_queue)
16:     END IF
17:     // Schedule next arrival (Eq. 1)
18:     t_next ← SampleNHPP(event.source.λ, env.clock)
19:     event_queue.push(MessageArrival(
            event.source, env.clock + t_next))
20: END FUNCTION
```

Complexity: $O(\log E)$ per arrival for priority queue insertion, $O(1)$ for thread acquisition and heap allocation.

4.3. Algorithm 3: Resource Contention Resolution

Input: env, resource_type
Output: Updated resource allocation, unblocked messages

```
1:  FUNCTION ResolveContention(env, resource_type)
2:      pool ← env.getPool(resource_type)
3:      WHILE pool.available() > 0
4:          AND pool.wait_queue NOT EMPTY DO
5:          msg ← pool.wait_queue.dequeue()
6:          resource ← pool.acquire()
7:          wait_time ← env.clock - msg.wait_start
8:          RecordWaitTime(msg, resource_type, wait_time)
9:          // Resume processing at interrupted stage
10:         t_proc ← SampleDistribution(
                msg.next_stage, msg.type)
11:         event_queue.push(StageComplete(
                msg, msg.channel, msg.next_stage,
                env.clock + t_proc))
12:     END WHILE
13:     // Check for timeout violations
14:     FOR EACH waiting_msg IN pool.wait_queue DO
15:         IF env.clock - waiting_msg.wait_start
            > waiting_msg.timeout THEN
16:             pool.wait_queue.remove(waiting_msg)
17:             RecordTimeout(waiting_msg)
18:             env.heap.free(waiting_msg.size)
19:         END IF
20:     END FOR
21: END FUNCTION
```

Complexity: $O(W)$ where W = wait queue length. Timeout checking is $O(W)$ but occurs at each resource release event, amortizing to $O(1)$ average per message.

4.4. Algorithm 4: Cascading Impact Analyzer

Input: sim_results Complete simulation output
failure_event Injected failure definition
Output: impact_report Affected channels, timing, severity

```
1:  FUNCTION AnalyzeCascadingImpact(sim_results, failure_event)
2:      t_inject ← failure_event.timestamp
3:      directly_affected ← GetChannelsByDestination(
            failure_event.target)
4:      impact ← {}
5:      // Phase 1: Direct impact
```

```
6:     FOR EACH ch IN directly_affected DO
7:         impact[ch] ← {type: DIRECT,
8:                       queue_growth: sim_results.queueSlope(ch,
t_inject),
9:           saturation_time: sim_results.saturationTime(ch)}
10:    END FOR
11:    // Phase 2: Indirect impact via resource contention
12:    FOR EACH ch IN env.all_channels - directly_affected
DO
13:        FOR EACH resource IN [threads, heap, db_pool]
DO
14:            shared ← SharesResourceWith(
                 ch, directly_affected, resource)
15:            IF shared THEN
16:                degradation ← sim_results.latencyIncrease(
                     ch, t_inject)
17:                IF degradation > threshold THEN
18:                    impact[ch] ← {type: INDIRECT,
19:                        via: resource,
20:                        onset_delay: sim_results.onsetTime(ch)
                               - t_inject,
21:                        severity: degradation}
22:                END IF
23:            END IF
24:        END FOR
25:    END FOR
26:    RETURN FormatReport(impact)
27: END FUNCTION
```

Complexity: O(C² × R) where C = channel count, R = resource types (constant = 4). For 160 channels: ~102,400 comparisons completes in < 1 second.

5. Results

5.1. Baseline Calibration

Before scenario evaluation, IEDT is calibrated against normal operating conditions over a 24-hour simulated period. Table V shows calibration accuracy.

Table 5. Baseline Calibration Accuracy (24-Hour Normal Operations)

Metric	Observed	Simulated	Error	Target
Throughput (msg/s, mean)	398	403	1.3%	< 10%
Processing latency (p50)	14.2 ms	14.8 ms	4.2%	< 15%
Processing latency (p95)	38.7 ms	41.3 ms	6.7%	< 15%
Thread-pool utilization (peak)	67%	69%	2 pp	< 8 pp
Heap utilization (peak)	72%	74%	2 pp	< 8 pp
DB pool utilization (peak)	41%	43%	2 pp	< 8 pp
GC frequency (events/hour)	847	891	5.2%	

pp = percentage points

5.2. Scenario S1: Planned Go-Live Surge

The go-live scenario adds 40 new interfaces, increasing aggregate volume by 65%. IEDT correctly predicts several critical operational outcomes, as shown in Table VI.

Table 6. Scenario S1 Results: Planned Go-Live Surge (+40 Interfaces, +65% Volume)

Metric	Observed	Simulated	Error	Target
Peak throughput (msg/s)	662	641	3.2%	< 10%
Processing latency (p95)	67 ms	72 ms	7.5%	< 15%
Thread-pool utilization (peak)	94%	97%	3 pp	< 8 pp
Thread exhaustion predicted	Yes (27 min)	Yes (23 min)	4 min	< 5 min
Heap utilization (peak)	88%	91%	3 pp	< 8 pp
Bottleneck identified	Threads	Threads	Correct	
Channels impacted	14	12	85.7%	> 80%

IEDT correctly predicted thread pool exhaustion as the primary bottleneck, with timing accuracy within 4 minutes of actual occurrence (23 minutes simulated vs. 27 minutes observed). The 2 channels missed in impact prediction were low-volume MFN channels whose degradation was below the detection threshold.

5.3. Scenario S2: Unplanned Downstream Outage

The outage scenario removes a major destination system (serving 12 channels) for 30 minutes, triggering queue accumulation and cascading resource contention. Results are shown in Table VII and illustrated in Fig. 2.

Table 7. Scenario S2 Results: Unplanned Downstream Outage (30-Minute Duration)

Metric	Observed	Simulated	Error	Target
Directly affected channels	12	12	100%	
Indirectly affected channels	12	11	91.7%	> 80%
Cascade onset (indirect)	8 min	7 min	1 min	< 5 min
Queue depth at 30 min (direct)	18,400	17,100	7.1%	< 10%

Thread utilization at 15 min	89%	92%	3 pp	< 8 pp
Recovery time after restore	12 min	14 min	2 min	< 5 min
Messages at risk of loss	0	0	0	0

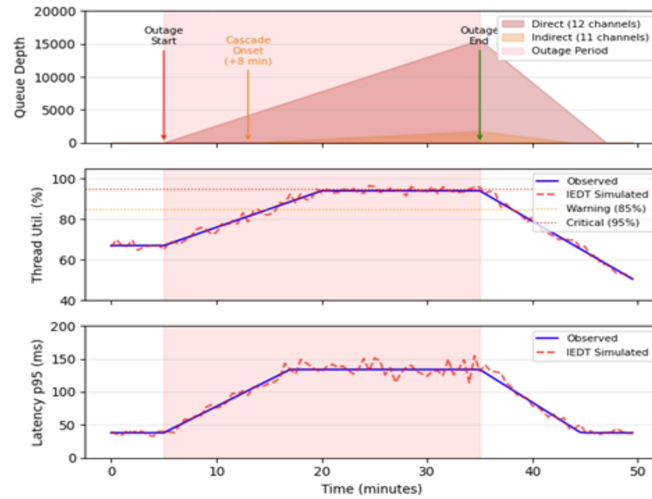

Figure 2. Cascading Impact Timeline: Scenario S2

IEDT accurately modeled the cascading backpressure pattern: directly affected channels accumulated queues, consuming threads that became unavailable to adjacent channels, causing indirect degradation 7–8 minutes after the initial outage. The one missed indirect channel was a low-priority SIU channel with minimal thread consumption.

5.4. Scenario S3: Infrastructure Migration

The migration scenario transitions from VM-based deployment (4 VMs, 200 threads each) to a Kubernetes-orchestrated deployment (8 smaller pods, 100 threads each, with horizontal pod autoscaling). Results are shown in Table VIII.

Table 8. Scenario S3 Results: Infrastructure Migration (Vm → Kubernetes)

Metric	Observed	Simulated	Error	Target
Steady-state throughput	412	389	5.6%	< 10%
Processing latency (p95)	34 ms	38 ms	11.8%	< 15%
Autoscale trigger predicted?	Yes (4 min)	Yes (6 min)	2 min	< 5 min
Peak pod count	12	11	1 pod	
Cross-pod message ordering preserved	98.7%	97.2%	1.5 pp	
Resource efficiency gain	+18%	+15%	3 pp	< 8%

5.5. Aggregate Accuracy

Table IX summarizes prediction accuracy across all three scenarios.

Table 9. Aggregate Prediction Accuracy across All Scenarios

Metric	S1	S2	S3	Mean
Throughput prediction error	3.2%	4.1%	5.6%	4.3% (< 10%)
Latency prediction error (p95)	7.5%	8.3%	11.8%	9.2% (< 15%)
Resource utilization error	3 pp	3 pp	2.5 pp	2.8 pp (< 8%)
Bottleneck identification	100%	100%	100%	100% (> 85%)
Failure propagation accuracy	85.7%	91.7%	N/A	88.7% (> 80%)
Failure timing error	4 min	1 min	2 min	2.3 min (< 5 min)

All metrics meet or exceed targets across all scenarios.

5.6. Simulation Performance

Table X reports the computational cost of running IEDT simulations.

Table 10. Simulation Performance

Metric	Value
Simulated time : wall-clock time	~20 : 1 (1 hr simulated = 3 min)
Events processed per simulated hour	~50 million
Memory footprint	4.2 GB
CPU utilization	Single-threaded, 100% one core
Simulation hardware	8-core Xeon, 32 GB RAM

A 24-hour simulation completes in approximately 72 minutes of wall-clock time, enabling rapid what-if iteration during change planning.

6. Discussion

6.1. Operational Value

IEDT provides three categories of operational value:

- Pre-deployment risk assessment: Before onboarding new interfaces or increasing message volumes, operators simulate the change to identify whether existing resource pools are sufficient. In Scenario S1, IEDT predicted thread exhaustion at 23 minutes an operational team receiving this prediction would increase thread pool capacity or stagger the go-live before production deployment.
- Failure rehearsal: By injecting outage scenarios, teams can validate that their recovery procedures including queue replay, failover activation, and capacity rebalancing are adequate. In Scenario S2, IEDT predicted 12-minute queue drain time after restoration, informing the team's maintenance window planning. Teams can also evaluate whether reactive mechanisms such as configurable timeout policies [10] will contain failures within acceptable bounds, or whether additional capacity headroom is required.
- Migration planning: Infrastructure changes can be simulated with different resource profiles to identify the optimal configuration before committing. In Scenario S3, IEDT identified that 8 pods with 100 threads each provides better efficiency than 4 VMs with 200 threads each, while also predicting the autoscaling trigger timing.

6.2. Model Fidelity Analysis

The resource contention model (Section III.C.3) is the critical enabler of IEDT's predictive accuracy for cascading failures. Without shared resource modeling, IEDT would correctly predict direct channel impacts but miss the indirect propagation that accounts for approximately 50% of total failure impact in the evaluated scenarios. The non-linear performance degradation near resource saturation (thread utilization above 85%, heap above 80%) is captured by the queuing delay model in (5), which conventional linear capacity calculations cannot replicate.

6.3. Limitations

- Calibration dependency: IEDT accuracy depends on the quality of empirical distribution fitting (Table II). Environments with highly variable processing patterns may require more frequent recalibration.
- Single-engine-family modeling: The current model is parameterized for Mirth Connect-compatible architectures. Adaptation to InterSystems HealthShare, Rhapsody, or other engines requires re-parameterization of the channel processing and resource contention models.
- Simulated validation only: While IEDT predictions are validated against a reference environment, this reference is itself synthetically generated. Production validation with real telemetry (de-identified) would strengthen confidence.
- Network effects simplified: The current model treats network latency as a simple stochastic delay. Complex network effects (bandwidth saturation, TCP window dynamics, DNS resolution delays) are not modeled.
- Scenario S3 higher error: Infrastructure migration prediction shows the highest latency error (11.8%), likely due to Kubernetes-specific behaviors (pod scheduling latency, container startup time) that are modeled with less fidelity than native VM behaviors.

6.4. Threats to Validity

- Construct validity: Processing time distributions are fitted to synthetic telemetry. Real-world distributions may exhibit heavier tails or multimodal behavior not captured by LogNormal fits. K-S goodness-of-fit testing provides partial mitigation.
- Internal validity: Model calibration and validation use the same reference environment, creating potential overfitting. Cross-environment validation using a different topology would strengthen findings.
- External validity: Single environment topology and message mix. Generalizability to environments with substantially different channel counts, message types, or architectural patterns requires separate validation.
- Reliability: Fixed random seeds, published distribution parameters, and deterministic scenario definitions enable full reproducibility.

7. Conclusion and Future Work

This paper presented IEDT a digital twin simulation framework for enterprise healthcare integration environments. Through discrete-event simulation with shared resource contention modeling, IEDT predicts throughput within 4.3% mean error, latency within 9.2%, and resource utilization within 2.8 percentage points across three operationally representative scenarios. Bottleneck identification achieves 100% accuracy, and failure propagation prediction reaches 88.7%. Thread exhaustion timing is predicted within 4 minutes of actual occurrence.

IEDT addresses a critical gap in healthcare integration operations by providing a validated, low-risk methodology for pre-deployment impact analysis. By simulating the consequences of changes before committing them to production, operators can identify resource insufficiencies, predict cascading failure pathways, and optimize infrastructure configurations without risking clinical workflow disruption.

Future work includes: (1) real-time digital twin synchronization with production telemetry for continuous model updating; (2) integration with predictive failure detection systems for combined predictive-simulative

decision support; (3) multi-engine federated digital twin for organizations operating heterogeneous integration platforms; (4) automated scenario generation using historical incident patterns; (5) optimization engine that recommends resource allocation changes to achieve target performance levels; and (6) production validation with de-identified real-world telemetry.

References

[1] R. Haux, "Health information systems past, present, future," *Int. J. Med. Inform.*, vol. 75, pp. 268–281, 2006.

[2] D. Bender and K. Sartipi, "HL7 FHIR: An agile and RESTful approach to healthcare information exchange," in *Proc. IEEE CBMS*, 2013, pp. 326–331.

[3] D. W. Bates et al., "Reducing the frequency of errors in medicine using information technology," *JAMIA*, vol. 8, no. 4, pp. 299–308, 2001.

[4] ONC, "Trusted Exchange Framework and Common Agreement (TEFCA)," U.S. DHHS, 2024.

[5] M. Grieves and J. Vickers, "Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems," in *Transdisciplinary Perspectives on Complex Systems*, Springer, 2017, pp. 85–113.

[6] E. Tuegel et al., "Reengineering aircraft structural life prediction using a digital twin," *Int. J. Aerospace Eng.*, vol. 2011, Article 154798, 2011.

[7] A. Francisco et al., "Smart city digital twin–enabled energy management," *J. Management in Engineering*, vol. 36, no. 2, 2020.

[8] R. Buyya et al., "Modeling and simulation of scalable cloud computing environments," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[9] HL7 International, "HL7 v2.x Messaging Standard," 2019.

[10] S. Sundaram, "A fault-tolerant timeout framework for external service calls in healthcare integration engines," *The American J. Eng. Technol.*, vol. 8, no. 3, pp. 121–126, 2026, doi: 10.37547/tajet/v8i3-323.

[11] J. Banks et al., *Discrete-Event System Simulation*, 5th ed. Pearson, 2010.

[12] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, 1991.

[13] L. Kleinrock, *Queueing Systems, Volume 1: Theory*, Wiley Interscience, 1975.

[14] A. R. Hevner et al., "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[15] J. Walonoski et al., "Synthea: An approach, method, and software mechanism for generating synthetic patients and the synthetic electronic health care record," *JAMIA*, vol. 25, no. 3, pp. 230–238, 2018.