Global Journal of Engineering and Technology Advances
eISSN: 2582-5003
Cross Ref DOI: 10.30574/gjeta
Journal homepage: https://gjeta.com/

(RESEARCH ARTICLE)

Check for updates

Regarding a testing framework designed to assess performance and compatibility in IoT systems

Nguyen Tai Tuyen [1,*] and Nguyen Quang Ngoc [2]

[1] *Faculty of Electronics Engineering 1, Posts and Telecommunications Institute of Technology, Hanoi, Vietnam.*
[2] *Faculty of Multimedia Communication, Posts and Telecommunications Institute of Technology, Hanoi, Vietnam.*

Global Journal of Engineering and Technology Advances, 2025, 25(02), 190-199

Publication history: Received 18 October 2025; revised on 24 November 2025; accepted on 26 November 2025

Article DOI: https://doi.org/10.30574/gjeta.2025.25.2.0335

Abstract

Modern Internet of Things (IoT) systems operate in highly heterogeneous environments with billions of devices, diverse protocols, and continuously evolving software versions. The primary challenge lies in ensuring performance under high load while maintaining cross-platform compatibility. Traditional testing methods address these factors independently, missing a critical class of errors that emerge only at their intersection-termed PC-Critical errors. This paper proposes PC-Testing, an integrated framework combining platform diversity modeling, distributed load simulation, and the Performance Degradation Index (PDI). Deployed on an Intelligent Transportation System (ITS) with 10,000 virtual devices using Locust, Appium, and NetEm, the framework revealed that the worst-case scenario (Android 8 with 300ms network latency) caused nonlinear degradation of 37.1× and 0.61% data loss-significantly worse than theoretical predictions. Experimental validation on Google Colab demonstrates the framework's effectiveness in detecting PC-Critical errors before production deployment.

Keywords: IoT testing; Performance Testing; Compatibility Testing; Cross-Platform; PDI; Latency; Data Integrity; Edge Computing

1. Introduction

1.1. Context and Motivation

The proliferation of Internet of Things (IoT) has transformed modern infrastructure across smart cities, telemedicine, Industry 4.0, and intelligent transportation. Statista projects over 75 billion connected IoT devices by 2030 [6], tripling from 2020 levels. However, this explosive growth introduces unprecedented heterogeneity in hardware (8-bit to 64-bit MCUs), operating systems (minimal embedded to full Android/Linux), communication protocols (MQTT, CoAP, HTTP/2, LwM2M, Zigbee, LoRaWAN), and continuous firmware updates [1], [2].

This diversity creates a highly heterogeneous environment where software quality assurance becomes the industry's most pressing challenge [3]. Traditional IoT architectures comprise three layers [1], [4]:

(1) Device/Sensor Layer for data collection with constrained resources, (2) Network/Edge Layer for preprocessing and gateway functions, and (3) Backend/Cloud Layer for long-term storage and analytics.

Each layer exhibits distinct bottlenecks under varying load conditions.

* Corresponding author: Nguyen Tai Tuyen.

1.2. The PC-Critical Problem

Among numerous IoT testing dimensions-usability, security, connectivity, scalability, energy efficiency -performance and cross-platform compatibility exhibit the deepest interdependence and pose the most severe system risks [7], [8]. Field deployments reveal that legacy firmware or outdated OS versions function normally under low load. However, when systems reach high concurrent load, inefficient packet processing in older devices triggers cascading bottlenecks at the Edge layer, ultimately collapsing the entire backend.

We define PC-Critical errors as defects manifesting exclusively when performance and compatibility factors intersect under high load. For instance, an Android 8.0 device on weak 4G (300ms latency) operates acceptably with 100 concurrent devices but causes 37.1× performance degradation at 5,000 devices, resulting in 0.61% data loss potentially fatal in Intelligent Transportation Systems (ITS).

1.3. Research Gap

Current testing approaches handle performance and compatibility in isolation:

- Performance testing tools (JMeter, Locust, Gatling) generate high load and measure throughput/latency but assume homogeneous clients, missing configuration-specific errors [9], [10].
- Compatibility testing tools (Appium, BrowserStack) validate applications across diverse platforms but operate at low concurrency (1–10 sessions), failing to detect high-load degradation [12].
- Mathematical models (queueing theory) assume homogeneous environments and cannot predict nonlinear degradation from platform diversity [13].
- Combinatorial Interaction Testing (CIT) reduces test cases but ignores performance factors [14], [15].

No existing work simultaneously achieves:

- multi-platform simulation,
- high concurrent load generation (>1000 clients), and
- quantification of compatibility-induced performance degradation.

1.4. Contributions

This paper presents four primary contributions:

4D Platform Diversity Model: A tuple representation

$$C_i = (\mathrm{OS}_k, \mathrm{AppVer}_n, \mathrm{NetW}_m, \mathrm{HardW}_l) \qquad (1)$$

reducing 5,000+ combinations to 5 high-risk configurations via 3-way CIT.

Hybrid Testing Architecture: Integration of Appium (client virtualization), Locust (distributed load), and NetEm (network emulation) enabling simultaneous multi-platform and high-load testing.

PDI Metric: Performance Degradation Index quantifying compatibility-induced performance loss through latency decomposition

$$\lambda_{i,j} = \lambda_B + \lambda_N(\mathrm{NetW}_m) + \lambda_I(C_i, L_j) \qquad (2)$$

Large-Scale Empirical Study: Deployment on real ITS with 10,000 virtual devices on Google Colab, discovering severe PC-Critical errors (PDI = 37.1×, 0.61% data loss).

2. Related work

2.1. Performance Testing

Load testing tools like JMeter [9], Locust [10], and Gatling [11] are industry standards for web/API performance testing. JMeter excels at HTTP/HTTPS but lacks native MQTT/CoAP support. Locust offers Python scripting flexibility but assumes client homogeneity. Gatling provides high throughput via Scala/Akka but lacks compatibility testing integration.

Mathematical approaches employ queueing theory (M/M/1, M/G/1) [13], Markov models, and fluid models. However, these assume uniform service rates, unsuitable for heterogeneous IoT environments. State explosion in Markov models and loss of per-platform detail in fluid models limit practical applicability.

2.2. Compatibility Testing

Appium [12] is the de facto standard for mobile automation (iOS/Android) with multi-language support. BrowserStack and Sauce Labs provide cloud-based testing on thousands of real devices but incur high costs (200+ USD/month) and prohibit high concurrency. Firebase Test Lab offers free Android testing but limits sessions to 15 minutes without performance profiling.

CIT techniques [14]–[16] demonstrate that pairwise (2-way) testing detects 50–90% of compatibility defects, while 3-way testing achieves 95–99% coverage. CIT reduces millions of combinations to hundreds but traditionally focuses on functional correctness, ignoring performance degradation.

2.3. Gap Analysis

Table 1 compares existing approaches with PC-Testing. No prior work simultaneously addresses multi-platform simulation, high concurrent load (>1000 clients), and quantitative performance degradation measurement.

Table 1 Comparison of IoT testing approaches

Method	Multi-Platform	High Load	PDI Metric
JMeter/Locust	×	✓	×
Appium/BrowserStack	✓	×	×
Queueing Theory	×	✓	~
CIT Testing	✓	×	×
PC-Testing	✓	✓	✓

3. PC-testing framework

3.1. Platform Diversity Modeling

3.1.1. 4D Configuration Space

Each IoT device configuration is represented as:

$$C_i = (\mathrm{OS}_k, \mathrm{AppVer}_n, \mathrm{NetW}_m, \mathrm{HardW}_l) \qquad (3)$$

where OS_k denotes operating system version, AppVer_n application version, NetW_m network quality, and HardW_l hardware specification. For ITS deployment:

$\mathrm{OS}_{\boldsymbol{k}}$: Android 8.0 (API 26, 2017) vs. Android 13 (API 33, 2022)

AppVer_n: v2.1.0 (synchronous REST) vs. v2.3.5 (WebSocket with retry)

$\mathrm{NetW}_{\boldsymbol{m}}$: 50 ms latency (good 4G/WiFi) vs. 300 ms (weak 4G)

$\mathrm{HardW}_{\boldsymbol{l}}$: 2 GB RAM vs. 4 GB RAM

3.1.2. Combinatorial Explosion and CIT

With 8 OS versions, 10 app versions, 8 network levels, and 6 hardware configs, the total combination space reaches:

$$8 \times 10 \times 8 \times 6 = 3{,}840$$

Including additional factors (firmware, battery, background apps) expands this beyond 5,000 combinations.

We apply 3-way CIT using IPOG (In-Parameter-Order-General) [14] to generate a minimal test suite ensuring every 3-tuple appears at least once. This reduces the space to 5 high-risk configurations (Table 2).

Table 2 High-risk configuration matrix

ID	OS	App	Net	Risk
C1	And 13	v2.3.5	50 ms	Low
C2	And 13	v2.1.0	50 ms	Medium
C3	And 8	v2.3.5	50 ms	Medium
C4	And 8	v2.1.0	300 ms	Critical
C5	And 13	v2.1.0	300 ms	High

C4 represents the worst-case scenario combining all adverse factors, while C1 serves as the ideal baseline.

3.2. Hybrid Testing Architecture

The PC-Testing architecture integrates three components:

Appium virtualizes heterogeneous clients with specified OS, app version, and hardware profiles. Standard Appium supports only 1–10 concurrent sessions; we overcome this via integration with Locust workers.

Locust generates distributed load through a master–worker topology. Each worker node runs a group of Appium clients with specific configurations, enabling scaling to 10,000+ concurrent devices.

NetEm (Linux Traffic Control) emulates network conditions including latency, jitter, packet loss, and bandwidth constraints. For C4, we configure:

tc qdisc add dev eth0 root netem \ delay 300ms 50ms loss 1% rate 2mbit

The architecture follows a master–worker pattern where the Locust master orchestrates multiple workers, each hosting Appium instances with NetEm-controlled network conditions. Workers communicate with an MQTT broker (HiveMQ), FastAPI backend, and MongoDB database replicating production ITS topology.

3.3. PDI Mathematical Model

3.3.1. Latency Decomposition

Total request latency decomposes into three components:

$$\lambda_{i,j} = \lambda_B + \lambda_N(\text{NetW}_m) + \lambda_I(C_i, L_j) \qquad (4)$$

λ_B(Base Latency): Backend processing time under ideal conditions, measured with single C1 client. Typical value: 20–30 ms.

$\lambda_N(\text{NetW}_m)$(Network Latency): Transmission delay dependent on network quality NetW_m, accurately emulated by NetEm. Values range from 5–10 ms (WiFi) to 300–500 ms (3G).

$\lambda_I(C_i, L_j)$(Incompatibility-induced Latency): The critical component emerging from interaction between suboptimal configuration $C_i$and high load L_j. This nonlinear term captures inefficient message queue handling in legacy OS, lack of smart retry in old app versions, RAM thrashing, and retransmission storms under weak network + high load.

3.3.2. PDI Definition

Performance Degradation Index quantifies compatibility-induced performance loss:

$$\mathrm{PDI}_{i,j} = \frac{\lambda_{i,j}}{\lambda_{\mathrm{baseline}}} \qquad (5)$$

where $\lambda_{\mathrm{baseline}}$ is measured under ideal conditions (typically 50 ms for C1 at 100 devices). PDI interpretation:

PDI = 1: No degradation

PDI < 5: Acceptable

$5 \leq \mathrm{PDI} < 10$: Warning - monitoring required

$10 \leq \mathrm{PDI} < 20$: Critical-intervention needed

PDI ≥ 20: Dangerous-system near unusable

For C4 at 5,000 devices: $\lambda_{C4,5000} = 1856$ms, yielding:

$$\mathrm{PDI}_{C4,5000} = \frac{1856}{50} = 37.1\times.$$

3.3.3. Prediction Model

To predict PDI for untested combinations:

$$\mathrm{PDI}_{i,j} = \alpha \cdot f_{\mathrm{OS}}(\mathrm{OS}_k) \cdot f_{\mathrm{App}}(\mathrm{AppVer}_n) \cdot f_{\mathrm{Net}}(\mathrm{NetW}_m) \cdot f_{\mathrm{Load}}(L_j) \qquad (6)$$

where $f_{\mathrm{OS}}, f_{\mathrm{App}}, f_{\mathrm{Net}}$are penalty factors for legacy components, and $f_{\mathrm{Load}}(L_j) = (L_j/L_{\mathrm{baseline}})^{1.2}$captures nonlinear scaling. For C4: $f_{\mathrm{OS}} = 1.5, f_{\mathrm{App}} = 1.3, f_{\mathrm{Net}} = 6.0$, yielding predicted PDI ≈ 34.0× (9.1% error vs. measured 37.1×).

4. experimental setup

4.1. System Under Test

We deploy PC-Testing on an Intelligent Transportation System (ITS) managing 5,000–10,000 vehicles. ITS represents an ideal case study due to:

- large scale,
- real-time requirements (latency > 1 s risks accidents),
- device diversity (modern buses to legacy taxis),
- variable network conditions (mobile coverage), and
- safety-critical consequences.

The ITS architecture follows standard three-layer IoT topology. Mobile apps (Android/iOS) collect GPS coordinates, speed, heading, and vehicle status every second. Data transmits via MQTT (QoS = 1) to a HiveMQ broker, then to FastAPI backend for processing and MongoDB storage. Backend performs congestion detection, travel time prediction, route optimization, and traffic light control.

4.2. Experimental Environment

Local Cluster (development): Intel Core i7-10700K, 32 GB RAM, Ubuntu 22.04, Docker + k3s. Limited to 2,000 concurrent devices.

Google Colab Pro (primary experiments): 12 vCPU (Intel Xeon @ 2.3 GHz), 52 GB RAM, Tesla T4 GPU (unused), 200 GB storage. Public internet with 80–220 ms jitter. Python 3.10, Ubuntu 20.04 in gVisor sandbox. Chosen for reproducibility (anyone can rerun experiments) and realistic cloud conditions (jitter, throttling, overhead).

4.3. Tools and Scenarios

Core tools:

- Locust 2.15.1 (load generation),
- Appium 2.0.1 (device virtualization),
- NetEm 2.6 (network emulation),
- Prometheus 2.45 (metrics),
- MongoDB 7.0 Atlas Free Tier (storage),
- HiveMQ Cloud (MQTT broker),
- FastAPI 0.104 (backend).

Test scenarios:

- Normal Load - 100/500/1000 devices at 1 msg/s for 60 s to establish baseline.
- Peak Hour - 2000/5000/10000 devices at 2 msg/s for 60 s to detect PC-Critical errors.
- Failure - worker crash mid-test to assess resilience.
- Stress - continuous load increase until system collapse.

Each device sends 250-byte JSON payloads containing device ID, timestamp, GPS coordinates, speed, heading, status, and configuration metadata. At 5,000 devices × 1 msg/s = 5,000 msg/s ≈ 1.25 MB/s throughput.

4.4. Metrics

Performance metrics:

Response Time $\lambda_{i,j}$(target < 500 ms),

Throughput (messages/s, target > 4000),

Error Rate (target < 1%),

Data Loss $D = (N_{\text{sent}} - N_{\text{received}})/N_{\text{sent}}$(target < 0.1%),

PDI (target < 10).

Resource metrics: CPU/RAM usage, network bandwidth, disk I/O, database connections.

QoS metrics: Availability (target 99.9%), reliability, consistency.

Data collection: 7 runs per configuration, 60 s per run, metrics sampled every 1 s. Outliers removed, mean and standard deviation calculated.

5. Results

5.1. Experimental Data

Table 3 compares simulation predictions with Google Colab measurements across 5 configurations at 5,000 devices.

Table 3 Simulation vs. Reality on Google Colab

Config	Simulation λ (ms)	Simulation PDI	Reality λ (ms)	Reality PDI	Error
C1	480	9.6×	512	10.2×	6.30%
C2	650	13.0×	687	13.7×	5.40%
C3	890	17.8×	945	18.9×	6.20%
C4	1700	34.0×	1856	37.1×	9.10%
C5	1120	22.4×	1203	24.1×	7.60%

C4 exhibits the most severe degradation. Detailed metrics for C4 at 5,000 devices: mean latency 1856 ms (+9.2% vs. simulation), p50 1620 ms, p95 3240 ms, p99 4850 ms, max 8120 ms. Actual throughput 6814 msg/s (17% below target due to retries). Total messages sent: 412,837; received: 410,312; data loss: 0.61% (2,525 messages).

5.2. Nonlinear Degradation Analysis

PDI scales nonlinearly with load for C4. At 100–500 devices: linear growth (10× to 15×). At 1000–2000: onset of nonlinearity (18× to 25×). At 3000–5000: explosive growth (30× to 37×). Beyond 2000 devices, multiple factors interact:

message queue saturation ⇒ timeouts ⇒ retries ⇒ increased load;

database connection pool exhaustion ⇒ waiting ⇒ increased latency;

near-full RAM ⇒ frequent garbage collection ⇒ CPU spikes;

network congestion ⇒ packet loss ⇒ retransmission storms.

Comparing configurations at 5000 devices:

- C1 (baseline) PDI = 10.2×;
- C2 (old app) PDI = 13.7× (+34%);
- C3 (old OS) PDI = 18.9× (+85%);
- C5 (weak network) PDI = 24.1× (+136%);
- C4 (worst case) PDI = 37.1× (+263%).

When multiple adverse factors combine, degradation multiplies rather than adds.

5.3. Data Loss Analysis

Data loss occurs only in C3 (0.12%), C4 (0.61%), and C5 (0.28%). For C4, 2,525 lost messages decompose:

MQTT layer (35%, 884 msgs): QoS = 1 PUBACK timeout due to broker overload.

Backend layer (45%, 1,136 msgs): garbage collection spikes (1.4–1.9 s) causing FastAPI unresponsiveness.

Database layer (20%, 505 msgs): MongoDB Atlas Free Tier write throttling and connection pool exhaustion.

Although 0.61% appears negligible, for ITS:

5,000 vehicles × 1 msg/s × 0.61% = 30.5 messages lost per second ≈ 1,830 per minute.

Consequences: lost GPS positions ⇒ undetected congestion ⇒ incorrect traffic light control ⇒ accident risk. Safety-critical systems cannot tolerate even 0.61% loss.

5.4. Root Cause Analysis

Model prediction (34.0×) underestimates reality (37.1×) by 9.1%. Four cloud-specific factors explain the discrepancy:

Network Jitter: Local NetEm provides stable 300 ms ± 5 ms. Google Colab public internet exhibits 80–220 ms jitter, causing inconsistent timeouts and excessive retries. Estimated impact: +156 ms for C4.

Garbage Collection: Python GC performs full collection every 30–40 s, spiking CPU to 100% for 1.4–1.9 s. During GC, FastAPI drops 7,000–9,500 requests at 5,000 msg/s rate, causing p99 latency > 5,000 ms. Mitigation: tune GC thresholds or migrate to Go/Rust (no GC pauses).

MongoDB Throttling: Atlas Free Tier limits write throughput to 100 ops/s. Beyond this, write concern “majority” delays from 50 ms to 500–1000 ms. Some writes rejected with “TooManyRequests” error. Estimated 505 messages lost (20% of total). Solution: upgrade to M10 cluster (≈57 USD/month) for 10× throughput.

gVisor Overhead: Colab’s gVisor sandbox intercepts all system calls for security, adding 7–11% overhead vs. native Docker. Particularly affects network/disk I/O. Estimated +120–180 ms for C4. Unavoidable platform limitation.

6. Discussion

PC-Testing successfully predicts nonlinear degradation trends with <10% error across all configurations, correctly identifying C4 as most dangerous. However, real-world cloud deployment proves more severe than simulation due to jitter, GC, throttling, and sandbox overhead. This confirms PC-Testing as an effective early warning tool requiring calibration for production cloud environments.

C4 represents the true tipping point where PDI > 30× renders the system nearly unusable and 0.61% data loss becomes unacceptable for safety-critical ITS. This defect class remains undetectable by traditional performance testing (homogeneous clients), traditional compatibility testing (low load), or local simulation (no cloud-specific factors). Only PC-Testing in realistic cloud environments reveals PC-Critical errors.

Latency decomposition for C4:

λ_B(base) = 25 ms (1.3%),

λ_N(network) = 300 ms (16.2%),

λ_I(incompatibility) = 1531 ms (82.5%).

The $\lambda_I$component dominates, explaining why PC-Critical errors are so severe-they scale nonlinearly with load and only manifest when compatibility and performance factors intersect.

6.1. Mitigation Strategies

Based on findings, we recommend:

- Back-pressure and Circuit Breaker: Implement rate limiting (10 msg/s per client) and circuit breaker pattern (open after 5 failures, 60 s timeout) to prevent overload.
- Exponential Retry with Dead Letter Queue (DLQ): Retry with exponential backoff (1 s, 2 s, 4 s); after 3 failures, route to DLQ for offline processing.
- Database Upgrade: Migrate from MongoDB Atlas Free Tier to M10 replica set, providing 10× write throughput and high availability.
- Time-Series Database: Replace MongoDB with TimescaleDB or InfluxDB, optimized for GPS time-series data with 5–10× better write performance and automatic retention policies.
- Runtime Migration: Replace Python with Go or Rust for services handling > 2,000 concurrent devices. Benchmarks show:

Python: 5,000 msg/s with 4,850 ms p99 latency and 8 GB memory;

Go: 15,000 msg/s with 120 ms p99 and 2 GB memory;

Rust: 25,000 msg/s with 45 ms p99 and 1 GB memory.

Graceful Degradation: Prioritize message processing by criticality:

- accident alerts and emergency vehicles,
- congestion detection and traffic light control,
- traffic statistics,
- historical analytics.

Drop priority 3–4 messages when load > 80%.

7. Conclusion and future work

7.1. Summary

This paper presents PC-Testing, the first integrated framework systematically addressing PC-Critical errors in large-scale heterogeneous IoT systems. Four key contributions:

- 4D platform diversity model reducing 5,000+ combinations to 5 high-risk configurations via CIT,
- hybrid architecture integrating Appium, Locust, and NetEm for simultaneous multi-platform and high-load testing,
- PDI metric quantifying compatibility-induced performance degradation with <10% prediction error,
- large-scale empirical validation on real ITS discovering severe PC-Critical errors (PDI = 37.1×, 0.61% data loss) invisible to traditional testing.

Experimental validation on Google Colab (November 18–19, 2025) with 10,000 virtual devices confirms PC-Testing's effectiveness. The framework successfully predicts nonlinear degradation trends, identifies worst-case configurations, and provides actionable mitigation strategies. Results demonstrate that PC-Critical errors-detectable only when performance and compatibility factors intersect under high load-pose serious risks to safety-critical IoT systems.

7.2. Limitations

Current limitations include:

- Scale-tested up to 10,000 devices vs. millions in real deployments;
- Diversity-Android only, MQTT only, 5 configurations vs. thousands possible;
- Environment-Google Colab only vs. AWS/Azure/GCP;
- Hardware-virtual devices only vs. real IoT hardware;
- Model-simple regression vs. AI/ML approaches.

7.3. Future Directions

AI/ML-Driven Prediction: Deploy LSTM models for real-time PDI forecasting, enabling 5–10 minute advance warnings before overload. Reinforcement learning agents could automatically adjust system parameters (scaling, timeout, rejection) to maintain PDI < 10.

6G and Digital Twin: Extend PC-Testing for 6G networks (1 Tbps, <1 ms latency) and digital twin applications requiring real-time physical–virtual synchronization. Expand to 6D model adding edge node type and 6G protocol dimensions. Simulate billions of devices in digital twins for predictive error detection.

Energy-Aware Testing: Develop E-PDI metric combining performance and energy consumption. PC-Critical errors waste energy through retries, timeouts, and prolonged connections. Green testing strategies could reduce IoT energy consumption 30–50%, contributing to Net Zero 2050 goals.

Additional Directions: Federated testing (distributed across edge nodes), chaos engineering for IoT (fault injection for resilience testing), continuous testing in CI/CD pipelines (automatic rejection of commits exceeding PDI threshold), blockchain-based test result verification (immutable audit trails).

7.4. Call to Action

We call on researchers to advance IoT testing techniques and extend PC-Testing to 5G/6G, digital twins, and metaverse applications. Developers should integrate PC-Testing into product development workflows, optimizing for worst-case rather than average-case scenarios. Organizations must invest in IoT testing infrastructure with diverse devices and network conditions. Regulatory bodies should establish mandatory IoT testing standards, especially for safety-critical applications like ITS and healthcare.

Compliance with ethical standards

Acknowledgments

This research was carried out at Faculty 1 of Posts and Telecommunication Institute of Technology. Experiments carried out on the Google Colab Pro platform. The research team wants to express their thanks to leadership Faculty of Electronic Engineering 1, Posts and Telecommunications Institute of Technology, We thank the open-source communities of Locust, Appium, and NetEm.

Disclosure of conflict of interest

No conflict of interest.

References

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[2] F. M. Al-Turjman, "Information-centric sensor networks for cognitive IoT: An overview," *Computer Communications*, vol. 101, pp. 85–95, 2018.

[3] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for Internet of Things: A survey," *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70–95, 2016.

[4] A. Botta, W. de Donato, V. Persico, and A. Pescape, "Integration of cloud computing and Internet of Things: A survey," *Future Generation Computer Systems*, vol. 56, pp. 684–700, 2016.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[6] Statista, "Internet of Things (IoT) connected devices worldwide 2030," 2025. [Online]. Available: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

[7] M. A. Razzaque et al., "Security and privacy in IoT: A survey," *IEEE Access*, vol. 5, pp. 10318–10329, 2017.

[8] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[9] Apache JMeter, "Apache JMeter – Performance testing tool," 2024. [Online]. Available: https://jmeter.apache.org/

[10] Locust, "Locust – A modern load testing framework," 2024. [Online]. Available: https://locust.io/

[11] Gatling, "Gatling – Load testing tool," 2024. [Online]. Available: https://gatling.io/

[12] Appium, "Appium: Mobile app automation made awesome," 2024. [Online]. Available: https://appium.io/

[13] D. Gross and C. M. Harris, *Fundamentals of Queueing Theory*, 4th ed. Wiley-Interscience, 2008.

[14] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr., "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[15] R. Brownlie, J. Prowse, and M. S. Phadke, "Robust testing of AT&T PMX/StarMAIL using OATS," *AT&T Technical Journal*, vol. 71, no. 3, pp. 41–47, 1992.

[16] K. Z. Zamli, F. Din, G. Kendall, and B. S. Ahmed, "An experimental study of hyper-heuristic selection and acceptance mechanism for combinatorial t-way test suite generation," *Information Sciences*, vol. 399, pp. 121–153, 2017.