

Article

STGen: A Lightweight Process-Based Testbed for Scalable IoT Protocol Evaluation with Physically Validated Synthetic Sensor and Anomaly Generation

Hasan M. A. Islam ^{1,*} , Md M. R. Maharaz ¹ , M. Georgiades ² , S. M. N. Shahriar ¹ , P. Akibuzzaman ¹ ,
N. R. Aurna ¹ , Md Masum ¹  and Riadul Islam ³ 

¹ Department of Computer Science and Engineering, East West University, Dhaka 1212, Bangladesh; rahmanmehraj627@gmail.com (M.M.R.M.); nafissahriar003@gmail.com (S.M.N.S.); pulok519@gmail.com (P.A.); nusrataurna07@gmail.com (N.R.A.); masummir773@gmail.com (M.M.)

² Department of Computer Science, Neapolis University Pafos, Pafos 8042, Cyprus; m.georgiades@nup.ac.cy

³ Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Baltimore, MD 21250, USA; riaduli@umbc.edu

* Correspondence: hasan.mahmood@ewubd.edu

Abstract

This paper introduces the Sensor Traffic Generator (STGen), a lightweight, pure-software testbed for evaluating IoT application- and transport-layer protocols at scale. Relative to existing software-based IoT evaluation platforms, STGen combines three design decisions that, to the best of our knowledge, no prior testbed offers together. Every emulated sensor node runs as an independent operating-system process using a real transport stack rather than a discrete-event model or a container. Sensor workloads are generated using physically grounded stochastic models calibrated against real deployment data. Experiments are specified in three independent tiers, IoT Protocols (N), Scenarios (M), and Networks (L), reducing configuration effort from a combinatorial $N \times M \times L$ problem to a linear $N + M + L$ workflow, with new protocols integrated by overriding a four-method abstract interface. STGen operates above OSI Layer 4 and therefore does not model PHY- or MAC-layer behavior, such as RF interference, CSMA/CA collision avoidance, or duty cycling. The sensor models are calibrated using 1,826,223 real-world readings from the Intel Berkeley Research Laboratory; for temperature, the synthetic stream matches the 37-day measurements of 54 Mica2Dot motes with a Kolmogorov–Smirnov D of 0.071 and a Jensen–Shannon divergence of 0.018, showing that STGen reproduces the statistical structure of real sensor data rather than only plausible values. By inverting these calibrated models, STGen also synthesizes labeled false-data-injection anomalies that are separable from normal traffic, with a receiver operating characteristic AUC of 0.898 for stealthy drift and 1.0 for hard physical range violations. In our experiments, STGen instantiates 6000 concurrently emulated sensor nodes on a commodity workstation in 1.02 s using 0.62 GB of memory (approximately 99 KB per node), which is more than two orders of magnitude below the per-node memory costs of container- and VM-based testbeds. STGen also exposes deployment-relevant behavior that controlled emulation alone may hide. Under live wide-area jitter, MQTT and CoAP exhibit different loss and latency patterns than those observed under uniformly degraded NetEm conditions, including MQTT reconnection storms. These results show that STGen provides a scalable and reproducible bridge between lightweight protocol emulation and practical deployment-oriented IoT protocol evaluation.



Academic Editors: Stefano Scanzio, Ramez Daoud, Jetmir Haxhibeqiri and Pedro Santos

Received: 1 July 2026

Revised: 28 July 2026

Accepted: 29 July 2026

Published: 3 August 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons](https://creativecommons.org/licenses/by/4.0/)

[Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: IoT testbed; protocol evaluation; synthetic sensor data; wireless sensor networks; network emulation; MQTT; CoAP; scalability

1. Introduction

Wireless Sensor Networks (WSNs) are distributed sensor networks that collect data from their surroundings that they broadcast within a range [1,2]. They are similar to ad hoc networks [3] in that they rely on wireless connectivity and spontaneous network formation to transport sensor data. A typical sensor node consists of a communication module, a processing unit, multiple sensor interfaces, and a power source (usually a battery or an embedded energy source) [4]. Energy limitations, along with the computational constraints of the sensor nodes, require efficient protocol design and evaluation to improve efficiency [5]. As network infrastructure simultaneously spans severely constrained low-power links (e.g., LoRaWAN at 250 bit/s) and high-throughput backhubs (e.g., 5G new radio (NR) at multi-Gbps), evaluation frameworks must characterize the application-layer and transport-layer protocol behavior throughout the entire link-capacity spectrum [6].

Protocol researchers and network engineers often use traffic generation tools to benchmark network performance and troubleshoot problems [7]. Although many frameworks and tools [8–13] have been proposed by research communities and the software industry, most of them lack support for lightweight and easily configurable traffic generation capabilities, as well as extensibility for Internet of Things (IoT)-specific application-layer protocols, particularly in controlled environments such as WSNs.

Several widely used network simulators have been proposed for evaluation of WSNs, each offering different trade-offs in scalability, modeling fidelity, and ease of use, as summarized in Table 1. Although network simulators provide users with control over the entire network stack, they create challenges in testing application-layer protocols. For example, OMNeT++ [14] requires advanced C++ proficiency, which presents a substantial barrier to entry for researchers. ns-3 [15] requires a complex setup and configuration through Python or C++. Cooja (Contiki) [16] provides a real operating-system-based simulation but suffers from limited scalability and a Java-based GUI. Although these tools are strong for network-layer research, the development of application-layer protocols can take weeks or even months to complete. STGen does not replace ns-3, OMNeT++, or Cooja, which reproduce PHY- and MAC-layer dynamics. In contrast, this study targets a complementary problem, i.e., the early-stage evaluation of application- and transport-layer protocols at thousands-of-nodes scale, where scalability and real protocol-stack behavior are the focus, rather than PHY/MAC fidelity.

Table 1. Common network simulators used for WSN evaluation.

Sim	Lang	Focus	Curve	Scale
ns-3 [15]	C++	Packet-level net sim and PHY/MAC models.	High	10 ⁴ +
OMNeT++ [14]	C++	Modular protocol simulation and component modeling.	V. High	3000+ ¹
Cooja [16]	Java	OS-level emulation (Contiki) and WSNs.	Mod.	<100

¹ <https://www.spec.org/cpu2006/Docs/471.omnetpp.html>, accessed on 28 July 2026.

Each existing platform family falls short of this target in a different way. Packet-level simulators (ns-3 and OMNeT++) and OS-level emulators (Cooja) provide PHY/MAC fidelity but impose high configuration effort and limited node counts for application-layer studies; physical testbeds such as FIT IoT-LAB provide real hardware but restrict node accessibility, reprogramming, and repeatability; container- and VM-based platforms scale to, at most, a few hundred nodes per host because of per-node virtualization overhead;

and digital-twin middleware such as Eclipse Ditto mirrors the device state for application development rather than measuring protocol performance under controlled network impairment. To the best of our knowledge, no existing platform combines (i) thousands of concurrently executing real L7/L4 protocol stacks in one commodity machine, (ii) statistically validated synthetic sensor workloads, and (iii) repeatable network-impairment and live-overlay experiments. STGen addresses this combination: a process-based node model at 99 KB marginal memory per node, sensor streams calibrated against 1.8 million real readings, and a three-tier configuration that reuses the same scenarios across protocols and network conditions. The scope of STGen provides an upper-bound performance baseline for layers above L4.

To facilitate protocol evaluation, this study proposes STGen, a lightweight, scenario-based testbed that emulates large-scale IoT-based wireless sensor network (WSN) environments. STGen instantiates thousands of virtual sensor nodes that produce high volumes of sensor traffic. It can be used to assess protocol performance in traditional IoT networks, where bandwidth is limited, and to evaluate the scalability, CPU performance, and end-to-end latency in environments representative of modern high-speed networks.

The STGen Core can handle a large number of virtual sensor nodes with low memory overhead, enabling large-scale application- and transport-layer experiments on a single machine. Moreover, STGen's scenario-based architecture uses extensible JSON scenarios that allow for the definition of domain-specific scenarios, such as smart healthcare and smart home scenarios. This approach allows researchers to closely recreate real-world conditions and test protocols across various application scenarios. Overall, the design facilitates the easy integration of simulated IoT environments and real-world application clients, which enables scalable, repeatable, and practical protocol research.

To evaluate STGen, we measured the end-to-end latency, packet loss, and host-resource scaling (CPU and memory) under varying load levels using a custom UDP-based publish-subscribe protocol [17,18]. The STGen Core acts as the designated sink node, servicing concurrent subscriptions and forwarding the captured network traffic to an ELK stack [19] for real-time latency and jitter extraction.

STGen enables researchers and developers to prototype device integration and evaluate the behavior of IoT application services, such as real-time monitoring, traffic shaping, and edge-level data analytics.

This work addresses four key research questions. (RQ1) investigates whether physically grounded synthetic sensor streams can faithfully reproduce the statistical characteristics of real-world sensor data, including marginal distributions, temporal dependencies, and cross-modality correlations. (RQ2) examines whether a process-based testbed architecture can achieve substantially greater scalability than virtual machine- and container-based approaches on commodity hardware while maintaining low per-node resource overhead. (RQ3) evaluates whether such a testbed can expose application-layer protocol behaviors and performance characteristics that are difficult to observe using conventional packet-level network simulators. (RQ4) investigates whether the same physically validated sensor models can generate labeled application-layer anomalous traffic that is statistically distinguishable from normal traffic, thereby supporting cybersecurity research and evaluation. The key contributions of this study are summarized as follows.

1. A physically validated synthetic sensor generator is proposed. Temperature, humidity, light, and voltage streams were produced by hardware-grounded models (Ornstein–Uhlenbeck thermal drift, PIR refractory logic, and MEMS noise floors) and validated against 1,826,223 real readings from the Intel Berkeley deployment, matching the marginal distribution (Kolmogorov–Smirnov $D = 0.071$ for temperature), temporal structure (autocorrelation within 0.035 of the measured curve at every tested lag),

and cross-modality coupling (temperature–humidity Pearson $r = -0.403$ against the measured -0.426).

2. A three-tier protocol/scenario/network decomposition that reduces experimental specification effort from a multiplicative $N \times M \times L$ to an additive $N + M + L$ is introduced.
3. A process-based emulation model is proposed that instantiates 6000 independent sensor-node processes on one 16 GB workstation at a marginal 99 KB per node, with each process holding a live transport stack and exposing the per-message latency, loss, and session state rather than an aggregate count.
4. We report evidence that the protocol behavior observed under emulated (NetEm) congestion does not carry over to a live deployment: across a three-site Tailscale/WireGuard overlay in Dhaka, the MQTT/CoAP loss ordering inverts relative to the emulated benchmark because steady-state uniform loss and bursty wide-area loss stress the two transports differently. A distributed deployment model in which software-defined nodes run across heterogeneous hosts and report to a central sink supports this measurement.
5. A reproducible evaluation workflow is developed. STGen includes named scripts to regenerate the reported tables and figures, statistical checks for protocol-latency comparisons, and a dual-archiving mechanism using local files and MongoDB to preserve the sensor streams during long-running experiments.
6. We propose an integrated adversarial engine that produces reproducible, labeled false-data-injection (FDI) datasets by perturbing STGen’s statistically validated sensor models. This feature generates configurable attack scenarios with complete ground-truth labels, enabling the systematic benchmarking of protocol resilience and anomaly-detection methods.

The remainder of the paper is organized as follows: Section 2 reviews existing IoT testbeds and identifies the gaps in scalability and architectural flexibility that STGen addresses. The core components and system architecture, including the physically validated sensor profiles and three-tier configuration model, are detailed in Section 3. Section 4 describes the simulation metrics, network emulation profiles, and hardware environment utilized for benchmarking. The evaluation methodology of the STGen framework is described in Section 5. The experimental findings and discussion are presented in Section 6. Section 7 discusses the extensibility of the framework and the integration workflow for the custom protocols. Section 8 presents the limitations of this study, threats to validity, and future directions. Finally, Section 9 concludes the paper.

2. State of the Art

In recent years, several experimental testbeds have been developed to support large-scale IoT environments for experimentation with IoT protocols and applications. For example, FIT IoT-LAB [20] and SmartSantander [21] have advanced this field by providing researchers with extensive sensor networks. However, these testbeds often have limitations in node accessibility and in their support for complex configurations. For instance, the main limitations of SmartSantander are its battery life and limited user reprogramming options. In contrast, FIT IoT-LAB focuses on providing access to bare metal but requires careful management of heterogeneous nodes and experimental setups.

SENS [13] is a modular simulator for wireless sensor networks that integrates sensor, environment, and network components. It enables realistic modeling of physical environments using tile-based propagation and supports application portability by allowing the same source code to run on simulated and real nodes. However, SENS is an older framework focused on component-based WSNs and provides limited support for evaluation of modern IoT application-layer protocols.

A separate lineage of WSN research tools achieves large node counts not through virtualization but through high-level abstraction, as surveyed by Sundani et al. [22]. Shawn [23] pioneered this philosophy: it models the effects of physical and MAC-layer phenomena rather than the phenomena themselves, which allows for simulations of more than 100,000 nodes on standard PC hardware but targets algorithmic and structural analysis rather than network-stack execution. Prowler [24] is a MATLAB-based, event-driven simulator for Berkeley MICA motes running TinyOS, modeling radio propagation, fading, and MAC-layer behavior to support simulation-based optimization of middleware services. Atarraya [25] is a discrete-event simulator for teaching and researching topology control algorithms, covering topology construction and maintenance, sensor data management, and communication routing. CupCarbon [26] supports the design, simulation, and 2D/3D visualization of WSNs for Smart City and IoT applications, including mobility, radio propagation, and interference modeling in urban environments. STGen shares the layer-abstraction premise of these tools but reaches scalability by a different mechanism: whereas they execute in-memory discrete-event models of protocol behavior, STGen runs actual application- and transport-layer (L7/L4) protocol stacks as operating-system processes and carries live traffic over real network interfaces and overlays, so the recorded latencies, losses, and session failures are produced by real protocol implementations rather than modeled ones.

TOSSIM [27] is a discrete-event simulator designed for use with the TinyOS operating system. It is high-fidelity in the sense that it builds TinyOS component graphs into the simulation infrastructure just in time, allowing the network behavior to be modeled at the bit level. However, TOSSIM is closely linked to the TinyOS-nesC environment and is therefore not suitable for analyzing general-purpose IoT application protocols, such as MQTT or CoAP, that run on a generic embedded C or Linux platform. In addition, its implementation model fails to account for CPU time-sharing or preemption, and performance is considerably reduced when handling network-intensive workloads.

Several frameworks have attempted to bridge the gap between simulation and emulation. IoTsim-Edge [28] focuses on the modeling of edge computing behaviors, whereas IOTier [29] offers a fully virtualized testbed. However, recent benchmarking by Bendaouch et al. [30] demonstrates that multi-tier (edge–fog–cloud) simulations incur substantial computational overhead, largely due to heavy virtualization dependencies.

Two adjacent recent lines of work should also be delineated from the scope of STGen. First, digital-twin middleware, as surveyed by Gil et al. [31], mirrors the state of physical devices into a queryable virtual representation (e.g., Eclipse Ditto) to support application development and integration. These platforms assume an existing device fleet and do not generate protocol workloads or apply controlled network impairment, which is precisely the experimental function provided by STGen. Second, recent protocol- and broker-benchmarking studies, such as the message-broker performance study by Paul et al. [32], characterize specific server implementations under load; STGen is a complementary infrastructure for such studies, supplying reproducible, statistically validated client-side workloads at thousands-of-nodes scale rather than benchmarking any single broker.

Gotham [8] and GothX [9] are tools for creating IoT datasets with both legitimate and malicious traffic. Gotham supports protocols such as MQTT and CoAP. GothX adds Kafka and SINETStream for improved IoT traffic generation. These tools allow for testing across different communication protocols and automate scenario execution. GothX offers greater flexibility with customizable topologies and automatic dataset labeling. They run each emulated IoT device as a Docker container, and each network router as a QEMU virtual machine, which costs approximately 37 MB per device and 470 MB per router

VM [9]. On a resource-constrained workstation, this limits the testbed to a few hundred nodes before memory is exhausted, and the shared virtualization layer adds resource contention between co-located guests.

PatIoT [11] was developed to offer a flexible IoT system testbed. This testbed scales from a physical setup to a simulated environment and supports various configurations. The framework is built on JUnit 5, which has been extended to support interoperability and integration tests. However, PatIoT requires users to manage complex event synchronization to explicitly specify this in the test code, increasing development effort and the complexity of simulating asynchronous IoT network conditions.

IoT-Flock [10] is an open-source framework that creates IoT traffic to help develop security solutions for smart homes by mimicking real-world IoT devices. It supports two IoT application-layer protocols: MQTT and CoAP. However, its native execution is currently non-functional, mandating VM deployment, which introduces substantial hypervisor overhead. Furthermore, it does not publish resource utilization metrics, making it difficult to assess its viability for large-scale benchmarking, where resource constraints are critical.

IoTGemini [12] generates synthetic IoT traffic that models device functions and network behaviors using a Packet Sequence GAN (PS-GAN). The framework focuses on statistical traffic modeling rather than stateful protocol execution. Consequently, it is not possible to confirm the functional integrity of application-layer protocols, including message delivery.

Table 2 compares STGen with representative physical testbeds, VM-based and hybrid platforms, simulators, and application-level frameworks. The comparison reveals a consistent gap: existing platforms either provide high-fidelity physical or packet-level modeling with limited ease of application-layer protocol experimentation or support IoT traffic generation at the cost of heavy virtualization overhead, limited scalability, weak network emulation support, or restricted protocol extensibility. These limitations motivate the need for a lightweight, scalable architecture that supports repeatable evaluations of application- and transport-layer IoT protocols on commodity hardware.

Table 2. Comparison of IoT protocol evaluation testbeds and frameworks. N/A = Not Applicable; N/R = Not Reported; Mod. = Moderate.

System	Type	Architecture/Scale	Protocol(s)	Boot Time	Mem	Net Emul.
<i>Physical Testbeds</i>						
FIT IoT-LAB [20]	Phy	Physical nodes (2700+ devices)	Hardware (any)	N/A	N/R	Ext. Tools
SmartSantander [21]	Phy	City-wide heterogeneous	Various	N/A	N/R	No
<i>VM-Based & Hybrid Testbeds</i>						
Gotham [8]	Hybrid	Docker + QEMU (~10 GB)	MQTT, CoAP	>20 min	~10 GB	Limited
GothX [9]	Hybrid	VM + Docker (~20 GB RAM)	MQTT, Kafka	~ 26 min	20 GB	Limited
IoTier [29]	Virt	Cluster-based	Generic	>5 min	High	Yes
<i>Application Frameworks & Simulators</i>						
IoT-Flock [10]	Frwkw	App-level (Host dependent)	MQTT, CoAP	Mod.	~400 MB	No
PatIoT [11]	Frwkw	JUnit-based (CI/CD)	Code-defined	N/R	High	No
IoTSim-Edge [28]	Sim	Edge/Cloud Modeling	Stream/MapRed	Mod.	N/R	Abstract
TOSSIM [27]	Sim	Discrete Event	TinyOS Only	Fast	Low	Built-in
SENS [13]	Sim	Generic Framework	Generic	Mod.	N/R	No
STGen (This Work)	Proc	Process-based (6000+ nodes)	Pluggable + FDI ‡	0.32–1.02 s †	0.62 GB	Yes

Bold entries indicate the proposed system (STGen), while ***bold italic*** entries denote category headers.

† Start-up scales linearly with node count: 0.32 s at 100 nodes, 0.44 s at 1000 nodes, and 1.02 s at 6000 nodes ($R^2 = 0.9995$). See Section 6.1. ‡ STGen additionally generates labeled false-data-injection (FDI) anomalies as departures from its physically validated baseline (Section 6.4).

3. System Architecture of STGen

STGen provides a flexible and extensible platform for simulating IoT environments and testing communication protocols. Its architecture is designed to support diverse scenarios while ensuring scalability and modularity.

3.1. Architectural Overview

The high-level architecture of STGen is shown in Figure 1. The STGen Core acts as middleware between the distributed sensor world and the client. In the distributed model, multiple STGen core instances can run concurrently, with each core coordinating numerous emulated sensors and client nodes. This structural decoupling enables easy testing of various protocols across different scenarios, such as smart homes, smart agriculture, or industrial IoT (IIoT) [33,34], without changing the core itself. During a run, the observability layer collects logs and performance metrics to enable comparison of protocol behavior across workloads and traffic patterns.

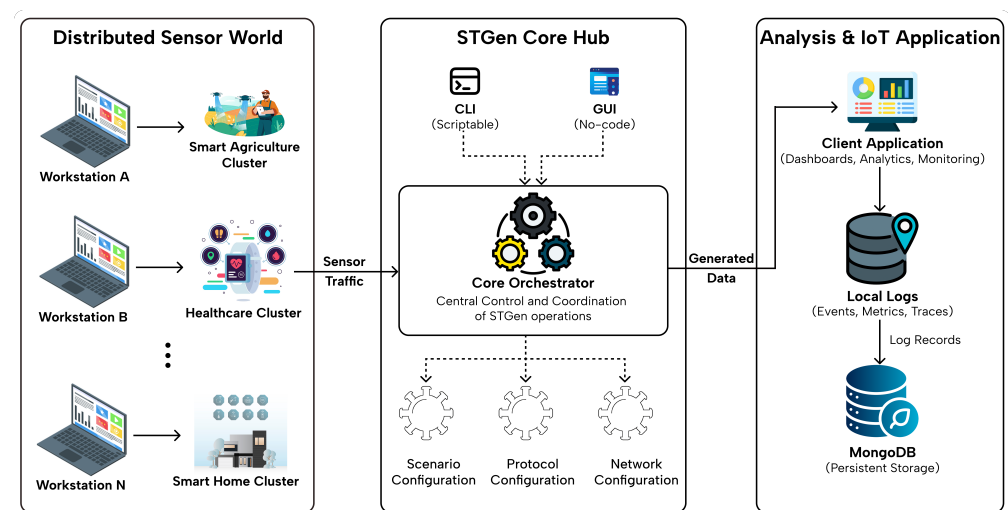


Figure 1. Overview of the STGen distributed IoT evaluation architecture. The framework bridges the distributed sensor world with analytical IoT applications via the central STGen core, facilitating a comprehensive protocol evaluation under realistic network conditions.

3.2. Core Components

The core components of the STGen testbed are described below.

3.2.1. Sensor Profile

Synthetic data is meaningless when it does not align with the physical constraints of the underlying architecture. To validate the synthetic data, STGen adheres to physical laws and hardware-specific dynamics through the following models. Table 3 summarizes the mathematical symbols, units, and parameters governing the STGen sensor profiles. The stochastic formulations themselves (Equations (1)–(5)) are established models adopted from the literature: the Ornstein–Uhlenbeck process [35], PIR refractory timing from the HC-SR501 datasheet [36], the MEMS noise-floor relation from the MPU-6050 datasheet [37], geodesic velocity clamping [38,39], and Weibull-distributed inter-arrival times [40]. The contribution of this work is not these equations but their integration: each model was parameterized from manufacturer datasheets, calibrated against the Intel Berkeley deployment (Section 6.3), and coordinated across thousands of concurrent nodes by the drift-compensated scheduler of Algorithm 1 (Equations (6)–(8)).

Table 3. Summary of mathematical variables and parameters used in STGen sensor profiles.

Category	Symbol	Unit/Type	Description
Thermal Dynamics [35]	T_t	°C / Float	Temperature value at time t
	θ	s^{-1}	Thermal relaxation rate (resistance to change)
	μ	°C	Long-term mean or ambient temperature
	dW_t	Brownian	Wiener process (Gaussian white noise)
Hardware Constraints [36,37]	T_{dwell}	s	Sensor high-state duration after activation
	T_{blocking}	s	Refractory period where sensor is unresponsive
	σ_{noise}	RMS	Calculated noise floor for vibration sensing
	NSD	$\mu g / \sqrt{\text{Hz}}$	Noise spectral density from device datasheet
	BW	Hz	Sensor bandwidth
Mobility & Kinematics [38,39]	v_{max}	m/s	Maximum allowable user or asset speed
	Δt	s	Time interval between generation steps
	Δ_{deg}	Degrees	Geodesic distance change in coordinates
Traffic & Control [40]	k	Shape ($k < 1$)	Weibull parameter controlling burstiness
	λ	Scale	Weibull parameter for characteristic IAT
	IAT	s	Inter-arrival time between packet bursts

Note: Bold rows indicate category headings used to group related mathematical variables and parameters.

- **Thermal Drift (Ornstein–Uhlenbeck Process):** Thermal inertia is observed in the temperature measurements. Randomly generated temperatures often produce unrealistic spikes that can trigger false alarms. To address this, STGen models the temperature evolution using the Ornstein–Uhlenbeck (OU) process [35]:

$$dT_t = \theta(\mu - T_t)dt + \sigma dW_t. \quad (1)$$

Here, θ is the thermal relaxation rate, μ is the long-term ambient mean, σ is the volatility that scales the random fluctuations, and dW_t is the Wiener process (Gaussian white noise).

- **Hardware Lockouts in PIR Sensors:** Standard simulations treat Passive Infrared (PIR) sensors as ideal instantaneous switches, whereas real modules (e.g., the HC-SR501) contain capacitors and internal logic that impose a refractory period [36]. After activation, the sensor remains in a high state for a dwell time (T_{dwell}), then becomes unresponsive for a blocking time (T_{blocking}). The minimum permissible trigger interval is defined as

$$\Delta t_{\text{trig}} \geq T_{\text{dwell}} + T_{\text{blocking}}. \quad (2)$$

This prevents unrealistic rapid bursts of motion events, ensuring that the packet rates and energy consumption reflect the limitations of the physical PIR hardware [41].

- **Noise-Floor Injection for Accelerometers:** Perfectly smooth signals are unrealistic in vibration sensing. Instead of adding arbitrary random noise, STGen derives the Root Mean Square (RMS) noise floor based on the MPU-6050 datasheet [37]:

$$\sigma_{\text{noise}} = \text{NSD} \times \sqrt{BW \times 1.6}, \quad (3)$$

where NSD is the noise spectral density and BW is the sensor bandwidth. The resulting noise is injected as a Gaussian process ($\mathcal{N}(0, \sigma_{\text{noise}}^2)$), ensuring that accelerometer readings exhibit realistic micro-vibrations.

- **Kinematic Clamping for GPS Sensors:** To maintain plausible mobility trajectories, we enforce a velocity-based spatial clamp similar to mobility-constrained models. Using geodesic approximations of 111 km per degree [38], STGen applies a velocity clamp as follows:

$$|\Delta_{\text{deg}}| \leq \frac{v_{\text{max}} \Delta t}{111,000}, \quad (4)$$

where $v_{\max}$ is the maximum allowable asset speed and Δt is the time step. This constraint prevents non-physical teleportation transitions from occurring between the coordinate points.

- **Traffic Shaping Using the Weibull Distribution:** IoT traffic seldom follows a Constant Bit-Rate (CBR) pattern. Instead, devices typically transmit data in irregular bursts, exhibiting self-similarity [42]. To model this, STGen generates inter-arrival times (IATs) using a Weibull distribution with a shape parameter of $k < 1$ (typically, $k \approx 0.8$) [40]:

$$f(t; \lambda, k) = \frac{k}{\lambda} \left(\frac{t}{\lambda} \right)^{k-1} e^{-(t/\lambda)^k}. \quad (5)$$

This distribution produces long, quiet periods followed by short bursts of packets, revealing congestion and queue overflows that are otherwise hidden in smooth simulations.

STGen follows the generation strategy outlined in Algorithm 1 to coordinate these physical models across multiple concurrent sensor nodes. The algorithm implements deterministic round-robin scheduling, ensuring that all N sensors transmit packets in a fair temporal sequence without head-of-line blocking. Critically, the system employs drift-compensated timing by maintaining a monotonic target wake time (t_{target}) and computing adaptive sleep intervals. The round-robin scheduler gives each sensor an effective physical update interval of

$$\Delta t_{\text{sensor}} = \tau N, \quad (6)$$

where τ is the global inter-transmission interval and N is the number of active sensor nodes. To avoid cumulative timing drift, STGen updates the target wake-up time after each transmission as

$$t_{\text{target}}^{(k+1)} = t_{\text{target}}^{(k)} + \tau, \quad (7)$$

and computes the adaptive sleep duration as

$$\delta_{\text{sleep}} = \max(0, t_{\text{target}} - t_{\text{now}}). \quad (8)$$

This ensures that variable processing delays do not accumulate across successive packet transmissions. Furthermore, it eliminates cumulative timing errors caused by variable processing delays, a technique reminiscent of slack stealing in real-time systems [43]. This approach ensures that inter-arrival times remain accurate, even under high CPU load. The effective physical timestep ($\Delta t = \tau \times N$) accounts for the round-robin period, ensuring that the updates to the temporal state correctly reflect the elapsed time between successive transmissions from the same device. Finally, as summarized in Table 3, all governing parameters are derived from manufacturer datasheets and validated against real-world sensor logs.

The computational cost of Algorithm 1 is bounded as follows. Each loop iteration performs a constant amount of work: one state update for the selected sensor (an OU step, refractory check, noise draw, or kinematic step, each requiring $O(1)$ time), one datagram transmission, and drift-compensation bookkeeping (one clock read, one addition, and one max). Therefore, generating M messages takes $O(M)$ time, regardless of how the messages are distributed across sensors. Memory is $O(N)$ in the number of sensors, with a constant per-sensor state record (current value, long-term mean, PIR timestamp); this constant is part of the 99 KB marginal per-node footprint measured in Section 6.1. The scheduling overhead of the round-robin discipline is the linear growth of the per-sensor revisit interval ($\Delta t_{\text{sensor}} = \tau N$): at a fixed global rate (τ), each sensor is updated less frequently as N grows, which the algorithm absorbs by scaling the physical timestep rather than by spawning per-sensor timers. The empirical scaling reported in Section 6.1, with

linear memory ($R^2 = 1.000$) and CPU rising in proportion to the aggregate offered load, is consistent with these bounds.

Algorithm 1 Physically Validated Sensor Data Generation with Drift Compensation

Require: Set of sensors $\mathcal{S} = \{S_1, \dots, S_N\}$, total messages M

Require: Inter-arrival time τ , OU parameters θ, μ, σ

Require: Hardware: $T_{\text{dwell}}, T_{\text{block}}$ (PIR), NSD (Accel), v_{max} (GPS)

Ensure: Time-synchronized stream of sensor packets

```

1:  $t_{\text{target}} \leftarrow \text{CurrentTime}()$  ▷ Drift compensation baseline
2:  $\Delta t \leftarrow \tau \times N$  ▷ Effective physical timestep per sensor
3: Initialize states  $\mathbf{x}$ , means  $\boldsymbol{\mu}$ , and timestamps  $\mathbf{t}^{\text{PIR}}$ 
4:  $k \leftarrow 0$ 
5: while  $k < M$  do
6:    $i \leftarrow k \bmod N$  ▷ Round-robin device selection
7:    $t_{\text{now}} \leftarrow \text{CurrentTime}()$ 
8:   ▷ Update Sensor Physics
9:   if  $S_i$  is Temperature then ▷ Ornstein-Uhlenbeck Process
10:     $z \sim \mathcal{N}(0, 1)$ 
11:     $x_i \leftarrow x_i + \theta(\mu_i - x_i)\Delta t + \sigma\sqrt{\Delta t} \cdot z$ 
12:   else if  $S_i$  is PIR then ▷ Refractory Logic
13:    if  $t_{\text{now}} - t_i^{\text{PIR}} > T_{\text{dwell}} + T_{\text{block}}$  then
14:       $x_i \leftarrow \text{Bernoulli}(p)$ 
15:      if  $x_i = \text{true}$  then  $t_i^{\text{PIR}} \leftarrow t_{\text{now}}$ 
16:    end if
17:   else
18:     $x_i \leftarrow \text{false}$ 
19:   end if
20:   else if  $S_i$  is Accelerometer then ▷ MEMS Noise Injection
21:     $\sigma_n \leftarrow \text{NSD} \cdot \sqrt{\text{BW} \cdot 1.6}$ 
22:     $x_i \leftarrow \mathbf{a} \sim \mathcal{N}(0, \sigma_n^2 \mathbf{I}_3)$ 
23:   else if  $S_i$  is GPS then ▷ Kinematic Random Walk
24:     $\phi \sim \mathcal{U}(0, 2\pi), \quad s \sim \mathcal{U}(0, 1)$ 
25:     $d_{\text{deg}} \leftarrow (s \cdot v_{\text{max}} \cdot \Delta t) / 111,000$  ▷ m/deg at equator
26:     $x_i \leftarrow x_i + d_{\text{deg}} \cdot [\cos \phi, \sin \phi]^T$ 
27:   end if
28:   Transmit UDP packet  $\langle \text{id}_i, t_{\text{now}}, k, x_i \rangle$ 
29:   ▷ Precise Drift Compensation
30:    $t_{\text{target}} \leftarrow t_{\text{target}} + \tau$ 
31:    $\delta_{\text{sleep}} \leftarrow \max(0, t_{\text{target}} - \text{CurrentTime}())$ 
32:   Sleep( $\delta_{\text{sleep}}$ )
33:    $k \leftarrow k + 1$ 
34: end while

```

3.2.2. STGen Core

The STGen core receives physically validated sensor streams generated by Algorithm 1 and serves as the central orchestration hub between sensor nodes and client applications. It includes a node registry that records the states of both clients and sensors, serving as a sink node that aggregates data from sensor nodes and facilitates delivery to client applications via the public Internet.

For experimental validation, the STGen core is equipped with a custom-designed, lightweight application-layer protocol based on the publish/subscribe (pub/sub) communication model. Upon receiving a sensor subscription request from a client, the protocol persistently records the client's state and subscription metadata. As multiple clients subscribe to one or more sensors, the core maintains a concurrent, scalable state registry for all active subscriptions, ensuring the efficient, timely dissemination of sensor data to relevant

subscribers as soon as it becomes available. In addition, the STGen core stores collected sensor data and application logs on the local hard disk and periodically persists sensor data in MongoDB [44] at runtime for further analysis. This architecture enables long-term evaluation of protocol performance under realistic IoT workloads while maintaining high-precision timing measurements through the drift-compensation mechanism implemented in the sensor generation layer.

3.2.3. Flexible Configuration

The STGen testbed can operate in both command-line interface (CLI) mode and Web server mode, offering flexible configuration options. User selections made through the Web interface are translated into API calls, which the Web server then converts into CLI commands to execute the required setup and control actions.

Overall, STGen is designed as a user-friendly tool with simple and intuitive configuration parameters. Its adaptability across a wide range of IoT research domains enables researchers and developers to model complex experimental settings efficiently and with minimal effort.

The backend service automates experimental workflows by exposing REST endpoints that map API requests directly to CLI executions. This abstraction eliminates the need for manual terminal interaction, streamlining the provisioning of sensors, clients, and core components while ensuring high reproducibility. When integrated with the frontend, the service utilizes WebSockets to stream real-time application logs and performance metrics directly to the browser, providing transparent insight into ongoing operations. Documentation for the REST API is handled through a comprehensive OpenAPI specification that is accessible via the built-in Swagger UI at /docs. This interface provides interactive testing capabilities and detailed endpoint documentation to facilitate third-party integration and versioning. For users who prefer low-level control, a complete reference of the underlying CLI commands is provided in Appendix A.

3.2.4. Client Application

Client applications on the STGen platform connect to the STGen core to interact indirectly with the sensor nodes. Clients initiate subscriptions by sending sensor-node requests to the STGen core, which manages state and data flow. This architectural separation of concerns ensures that the core is solely responsible for handling the complexities of sensor registration, state management, and data dissemination. At the same time, clients remain focused on consuming and processing sensor data. This architecture accommodates numerous clients interfacing with the same STGen core.

3.3. STGen as a Hybrid Network Model

In STGen, the emulated Wireless Sensor Network (WSN) nodes are not constrained to a single physical machine. These software-defined nodes can be distributed across multiple systems and connected to the STGen core via wired or wireless channels. This architectural flexibility allows for the creation of spatially separated WSN clusters while keeping them synchronized with the core. The STGen core is isolated from the host system of the emulated nodes, allowing for a hybrid network model that combines wired and wireless communication modalities. Additionally, as shown in Figure 1, STGen provides a testbed that serves as a powerful tool for traffic generation and protocol evaluation in hybrid IoT networks. The testbed emulates thousands of resource-constrained sensors and actuators [45]. In WSNs, sensor nodes are typically placed in a private network area. Therefore, direct communication with external systems is usually done via a designated sink node. The proposed hybrid network model assigns a physical machine called the

STGen core as the sink node and as an offloading node for the emulated sensor nodes to offload their computational and communication loads.

3.4. Three-Tier Configuration Model

As highlighted in Table 2, existing frameworks often suffer from high initialization times and complex deployment procedures. To address this, STGen introduces a configuration model that partitions experiments into three independent tiers. This structural decoupling ensures that the protocol logic remains distinct from application scenarios and environmental constraints, as illustrated in Figure 2.

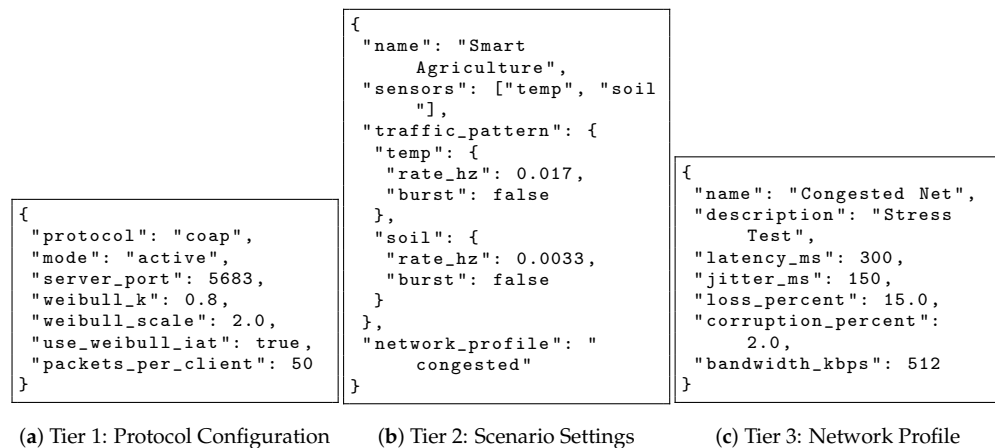


Figure 2. JSON configuration examples demonstrating STGen’s modular three-tier architecture: (a) Protocol configuration defines protocol mechanics; (b) scenario settings define the application context and link to the selected network profile; and (c) network profile defines environmental constraints such as latency, jitter, packet loss, corruption, and bandwidth.

- **Tier 1: Protocol Configuration** This tier separates protocol-specific parameters from the rest of the system. It controls how data is delivered (e.g., setting MQTT quality of service levels) without affecting how the data is actually generated. Figure 2a illustrates this separation, defining a CoAP configuration that specifies a Weibull distribution for inter-arrival times independently of the specific sensors being simulated.
- **Tier 2: Scenario Configuration** This tier defines the application context. It dictates the active sensor types, node counts, and specific traffic patterns. As shown in Figure 2b, a smart agriculture scenario specifies low-bandwidth temperature and soil moisture sensors. Crucially, this file acts as the bridge, referencing the target network profile to link the tiers at runtime.
- **Tier 3: Network Profile** This layer controls the environment. It applies specific link characteristics, such as latency, jitter, and packet loss, to simulate real-world network instability. Figure 2c demonstrates a congested network profile, imposing high latency (300 ms) and significant packet loss (15%) to stress-test the chosen protocol.

These tiers merge at runtime to form a complete executable experiment. This approach significantly reduces the configuration workload by shifting the effort from a multiplicative to an additive factor. For instance, evaluating four protocols across six scenarios under five network conditions requires defining only 15 source files ($4 + 6 + 5$) rather than managing 120 separate monolithic configuration files ($4 \times 6 \times 5$).

This reduction is a direct counting argument rather than a derived theoretical model, and it holds under two assumptions enforced by the tier design. First, an experiment is fully specified by a triple (protocol, scenario, and network profile), so a monolithic scheme requires one file per combination: $C_{\text{monolithic}} = NML$. In comparison, the tiered scheme requires one file per tier element ($C_{\text{STGen}} = N + M + L$), which is composed at

runtime. Second, the tiers must be independent: no parameter in one tier may depend on the selection in another. STGen's configuration schema satisfies this by construction because a scenario references a network profile by name only and protocol files carry no scenario- or network-specific fields. If cross-tier coupling were introduced (for example, a protocol parameter tuned per network profile), the affected combinations would, again, require dedicated files, and the additive bound would no longer apply.

3.5. Dual-Archiving for Redundancy and Scalability

Because the STGen core is ephemeral, it is essential to aggregate data from different sensor nodes into a storage system. For this, the STGen architecture performs two distinct tasks: first, distributing the data to MongoDB for efficient retrieval and analysis and second, ensuring that the sensor data is aggregated locally. This dual-archiving strategy ensures redundancy and facilitates real-time access and subsequent temporal analysis. The network traffic log is indexed and stored in Elasticsearch for real-time visualization and meaningful insights.

3.6. Real-Time Network-Log Analysis

To evaluate the performance of the underlying protocols, network performance analysis is crucial during client–sensor interactions through the STGen middleware. Capture of real-time network traffic on the designated STGen middleware ports is achieved with TShark, the command-line version of Wireshark. However, raw traffic data lacks structure and requires transformation for meaningful analysis, necessitating the identification of key performance metrics, including latency, jitter, packet loss, retransmissions, and protocol overhead. To address this, Logstash preprocesses and transforms raw traffic data in real time, extracting relevant features before forwarding it to Elasticsearch, where it is indexed and stored for efficient querying and analysis [46].

The STGen core node captures this network traffic by extracting real-time data from specific ports and outputting it in JSON format, making it easier to feed into Elasticsearch for further processing. Researchers evaluating communication protocols can leverage this indexed data to analyze protocol performance under various conditions. By integrating Elasticsearch with Kibana, researchers gain access to dynamic, real-time visualizations that enable effective benchmarking of network behaviors. Kibana's dashboards, time-series charts, and heatmaps provide insights into latency trends, traffic congestion points, and packet-size distributions across sensor nodes. By tailoring these visualizations to emphasize specific network performance metrics, researchers can extract actionable insights from STGen's network traffic, optimizing both protocol design and middleware efficiency. Figure 3 illustrates real-time delay statistics visualized in Kibana, showcasing the effectiveness of this TShark and ELK workflow in STGen's testbed analysis.

Interaction Layers and User Interfaces

STGen offers functionality via three major interfaces.

1. **Command-Line Interface (CLI):** STGen primarily offers a CLI-based interface for configuring system parameters. Users can launch the entire testbed and customize its behavior through command-line arguments.
2. **Web-Based Graphical User Interface:** The Web interface simplifies configuration management. It connects directly to the backend services and converts user input into standard API calls.
3. **REST API on OpenAPI Standards:** The main functionality is provided via a RESTful API. The standardized interface allows for interactive testing.

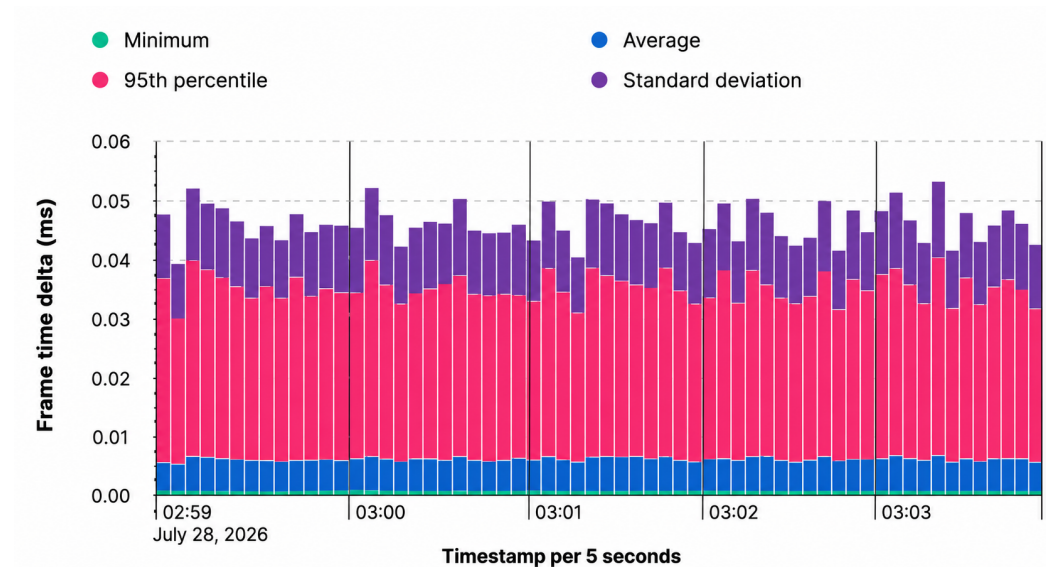


Figure 3. Real-time statistics of network traffic delay in STGen Traffic Analyzer. This figure presents the minimum, average, standard deviation, and 95th percentile of frame-time delta (delay) over time, visualized with the ELK stack for real-time traffic analysis.

4. Experimental Configuration

We conducted several experiments across different scenarios. This section lists their hardware, measurement methods, and network configurations.

4.1. Hardware Environment

The hardware specifications of the workstations used in this study are summarized in Table 4. Multiple heterogeneous machines were intentionally selected to evaluate STGen under both controlled and non-uniform execution environments.

- **Workstation A (Standard Testbench and Distributed Core):** Served as the primary platform for single-machine scalability analysis and functioned as the central orchestrator in the distributed setup.
- **Workstations B and C (Distributed Clients):** Geographically dispersed hosts located in Old Dhaka and Demra, respectively. These nodes were used to emulate external sensor clients operating under stochastic network conditions over a public internet overlay.

Table 4. Hardware specifications of experimental workstations.

Feature	Workstation A	Workstation B	Workstation C
Primary Role	Scalability & Protocols	Distributed Node 1	Distributed Node 2
CPU	AMD Ryzen 5 5600 G	Intel Core i3-3110M	Intel Core i5-10210U
RAM	16 GB DDR4	8 GB DDR3	8 GB DDR4
OS	Ubuntu 24.10	Linux Mint 22.2	Kali GNU/Linux Rolling
Location	Dhaka (23.74° N, 90.43° E)	Dhaka (23.72° N, 90.43° E)	Dhaka (23.68° N, 90.43° E)

The hardware selection aligns with the goals of the two experimental classes. Workstation A, a mid-range desktop (6-core Ryzen 5 5600 G, 16 GB RAM), represents the commodity hardware class that STGen targets. Hence, all scalability and protocol benchmarks were conducted on this single machine under identical conditions. Workstations B and C were deliberately chosen to be weaker and mutually heterogeneous (a 2012-class dual-core i3 with DDR3 and a 2019-class quad-core i5): the distributed experiments are meant to exercise STGen under non-uniform, deployment-like conditions, not to compare absolute perfor-

mance across hosts. Therefore, hardware heterogeneity affects the distributed results only through the realism it adds; no cross-host latency or throughput comparisons are drawn between machines with different specifications. The dependence of the single-machine coefficients on Workstation A's CPU, memory subsystem, and scheduler is analyzed as an internal-validity threat in Section 8.

4.2. Measurement Methodology

To ensure consistent comparison across scalability and protocol experiments, we measured both system-level resource usage and communication-level performance using the following metrics.

- **Startup Time:** Measured using `time.perf_counter()` with 1 ms precision and defined as the elapsed time between the orchestration launch command and the initialization of all client threads.
- **Memory Usage:** Resident Set Size (RSS) measured via `psutil` every 0.1 seconds and aggregated across the parent process and all child processes.
- **CPU Load:** Instantaneous CPU utilization sampled at 10 Hz (0.1 s interval), with the reported value representing the peak utilization observed during the test window.
- **Throughput and Latency:** Throughput is reported as the message rate from valid server-side receipts over the test duration. A bandwidth figure in Mbps is derived only for the custom UDP protocol, whose frame is a fixed 112 bytes (12-byte STGen header plus 100-byte payload); we do not report a byte-rate for MQTT or CoAP because their header and option lengths vary per message (topic length, token length, and quality-of-service flags), so a single frame size would not represent them. Latency is computed as Δt between the client-side send timestamp and the server-side write timestamp.

4.3. Network Emulation Profiles

All protocol comparisons were conducted on Workstation A using the Linux NetEm tool [47] to inject artificial network impairments. NetEm is part of the Linux `tc` subsystem, so this impairment injection runs only on Linux hosts; all single-machine and distributed runs reported in this article used Linux, and cross-platform support via WSL2 is left to future work (Section 8.3). Table 5 lists the NetEm network-condition presets.

Table 5. Network-condition profiles.

Profile	Delay (ms)	Jitter (ms)	Loss (%)	Bandwidth (Kbps)
Wi-Fi	20	5	0.5	54,000
4G/LTE	70	10	1.2	15,000
LoRaWAN	600	40	2.5	50
Congested	150	30	5.0	1000
Perfect	1	0	0	Unbounded

4.4. Simulation Parameter Description

This subsection details the configuration parameters required for the STGen testbed. To facilitate a modular and reproducible experimental design, STGen utilizes a tiered configuration schema that decouples communication logic from environmental constraints.

As summarized in Table 6, the configuration is categorized into three functional tiers: a protocol tier, scenario tier, and network tier. These parameters ensure the three-tier configuration model of STGen.

Table 6. Configuration parameters for STGen simulation setups.

Parameter	Description
<i>Protocol Tier</i>	
Protocol	IoT protocol to evaluate (MQTT, CoAP, SRTP, Custom UDP)
Mode	active (traffic generation) or passive (traffic replay)
Server IP/Port	Networking coordinates for the STGen Core or broker
<i>Scenario Tier</i>	
Num. Clients	Total number of simulated concurrent sensor nodes
Duration	Total experiment execution time in seconds
Sensors	List of sensor types (temp, humidity, gps, camera)
Role	core (orchestrator) or sensor (generator)
<i>Network Tier</i>	
Profile	Presets: WiFi, 4G, LoRaWAN, Congested, Perfect
Network Stats	Manual overrides for Latency (ms), Loss (%), and Jitter

Note: Italicized rows indicate category groupings for the parameters.

Furthermore, to maintain consistency across comparative studies, STGen provides pre-configured network-condition profiles. As detailed in Table 5, these profiles (e.g., Wi-Fi, 4G/LTE, and LoRaWAN) define specific parameters for delay, jitter, packet loss, and bandwidth. This structured approach lets the platform run in both local and distributed environments, allowing researchers to evaluate protocol performance under realistic, varying levels of network stress.

5. Evaluating the STGen Framework

We evaluated STGen along three axes to validate its design and practical utility: goal-based, process-based, and outcome-based evaluations.

- **Goal-based evaluation** examines whether the implemented features align with the objectives established during the requirements phase. For STGen, this meant verifying the physical sensor’s accuracy, timing precision, and ability to support researchers and enthusiasts in verifying protocols during early development stages.
- **Process-based evaluation** assesses the system’s quality and architectural implementation, specifically validating the modularity of the framework’s architecture.
- **Outcome-based evaluation** examines practical use: whether the system supports valid, reproducible IoT protocol evaluation.

Figure 4 shows how the three evaluation axes feed back into the design. The goal-based check is the sensor-fidelity result from Section 6.3, in which the stochastic models reproduce the statistical characteristics of the Intel Berkeley dataset. The process-based check evaluates the framework’s modular design via the protocol integrations described in Section 7. The same separation of concerns decouples sensor generation from transmission, allowing new sensor types to be added independently of the communication layer.

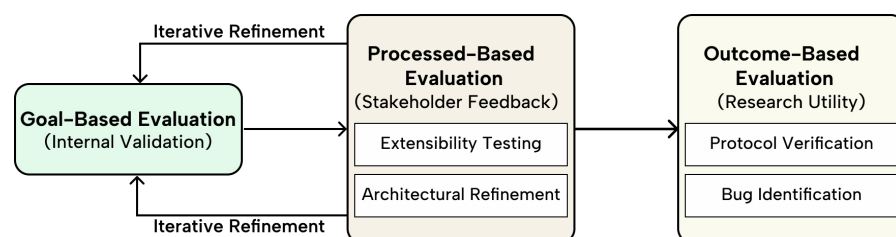


Figure 4. STGen evaluation life cycle, illustrating the iterative flow between internal validation (goal-based), stakeholder feedback (process-based), and research utility (outcome-based).

The outcome-based evaluation demonstrates STGen's practical utility. STGen allows researchers to verify an IoT protocol implementation and reproduce a protocol bug without buying hardware. The MQTT and CoAP studies reported in Section 6 are examples of that workflow.

6. Experimental Results and Discussion

This section reports six experiments, each stated as a question before its measurements. Section 6.1 asks how single-machine resource use scales with node count. Section 6.2 asks whether per-protocol latency stays bounded as concurrency rises and where each protocol's client stack saturates. Section 6.3 asks whether the synthetic sensor streams match field data. Section 6.4 asks whether the same validated models generate labeled anomalies that are statistically separable from normal traffic. Section 6.5 asks how MQTT and CoAP behave under emulated congestion. The distributed study asks how the same two protocols behave over a live wide-area overlay rather than an emulated link.

6.1. Single-Machine Scalability

This experiment measures how STGen's resource usage scales as the entire testbed runs on a single workstation (Workstation A). It answers two questions:

1. How do startup time, memory, throughput, and CPU grow with the number of sensor nodes?
2. What is the single-machine node ceiling on this hardware, and which resource limits it?

This experiment uses STGen's native node model, in which every emulated sensor runs as an independent operating-system process launched by the orchestrator. It measures the cost of instantiating and holding n such processes rather than any one protocol's broker or acknowledgment path (measured separately in Section 6.5). Each node is generated at a realistic 1 Hz; the footprint is rate-independent because per-node memory is the process image, not the traffic. We swept the node count from 100 to 6000 and recorded the startup time, memory usage, and CPU usage for the entire process tree on Workstation A. Memory is reported as the Proportional Set Size (PSS), which charges shared library and binary pages once across the identical node processes, whereas the summed resident set size would double-count them.

6.1.1. Scalability Analysis

Table 7 and Figure 5 report the process-based scaling sweep from 100 to 6000 nodes on Workstation A. Every point instantiates that many independent OS processes; the live-process count matched the target exactly at every scale. Three observations characterize the resource scaling and the node ceiling.

- **Startup time:** Initialization time grows linearly with node count. An ordinary least-squares fit to the seven measured points reported in Table 7 gives Equation (9):

$$T(n) \approx 1.17 \times 10^{-4} n + 0.31 \quad (R^2 = 0.9995), \quad (9)$$

where n is the node count and $T(n)$ is the startup time in seconds. The 95% confidence intervals on the fitted coefficients are $[1.14, 1.20] \times 10^{-4}$ s/node for the slope and $[0.306, 0.323]$ s for the intercept ($n = 7$, t -based). All 6000 node processes are spawned and running in 1.02 s.

- **Memory footprint:** Memory scales linearly, with no super-linear term indicating a leak. The least-squares fit to the summed PSS is Equation (10):

$$M(n) \approx 98.7 n + 29,000 \text{ [KB]} \quad (R^2 = 1.000). \quad (10)$$

The 95% confidence intervals are [98.5,99.0] KB/node for the slope and [28.6,29.7] MB for the intercept. For both regressions, the standardized residuals stay within ± 1.5 , with no systematic pattern across the sweep, and a Shapiro–Wilk test does not reject residual normality (startup $p = 0.48$, memory $p = 0.66$). These fits are interpolations over the measured 100–6000-node range; we do not extrapolate them beyond 6000 nodes, where per-process kernel structures and scheduler overhead can introduce non-linear terms that the sweep does not observe. The marginal cost is 98.7 KB per node, and the intercept is a 29 MB fixed base; both coefficients are specific to this CPU, C library, and Linux scheduler (Section 8). The per-node figure in Table 7 falls from 387 to 104 KB as shared C-runtime and binary pages amortize across more processes; the marginal 99 KB is the true incremental cost of one more node.

- **CPU and node ceiling:** The CPU of the process tree rises with the aggregate offered load, from 4% at 100 nodes to 72% at 6000 (Figure 5). Memory is not the binding constraint: 6000 nodes use 622 MB, corresponding to under 4% of the 16 GB host. We do not claim a specific higher ceiling because we did not run beyond 6000 nodes; per-process kernel structures and scheduler overhead impose a real upper bound, and determining it is left for future work.

Table 7. Process-based single-machine resource scaling on Workstation A (16 GB RAM, 6 cores). Memory is the sum of the proportional set size of the node process tree.

Nodes	Startup (s)	Memory PSS (MB)	Per-Node (KB)	CPU (%)
100	0.32	39	387	4
250	0.35	54	216	7
500	0.37	79	158	13
1000	0.44	128	128	22
2000	0.55	226	113	36
3000	0.66	325	108	48
6000	1.02	622	104	72

Each node is an independent OS process generating at 1 Hz. Per-node PSS falls as shared library pages amortize across more processes; the marginal cost is 99 KB/node. CPU is the process tree's aggregate as a fraction of one core (100% = one full core).

6.1.2. What Each Node Process Holds

To show that the 99 KB marginal cost is a working node rather than an idle placeholder, Table 8 decomposes one node's resident memory, read from `/proc/<pid>/smaps`. Each process is a live UDP client with its own socket, send buffer, and sequence and timing state. Its summed PSS is 118 KB, of which about 19 KB is a shared, read-only C-library and binary text that amortizes as more nodes launch, which is why the marginal slope of Equation (10) is 99 KB rather than 118 KB. Kernel socket buffers are not charged to PSS, so the figure is a userspace lower bound.

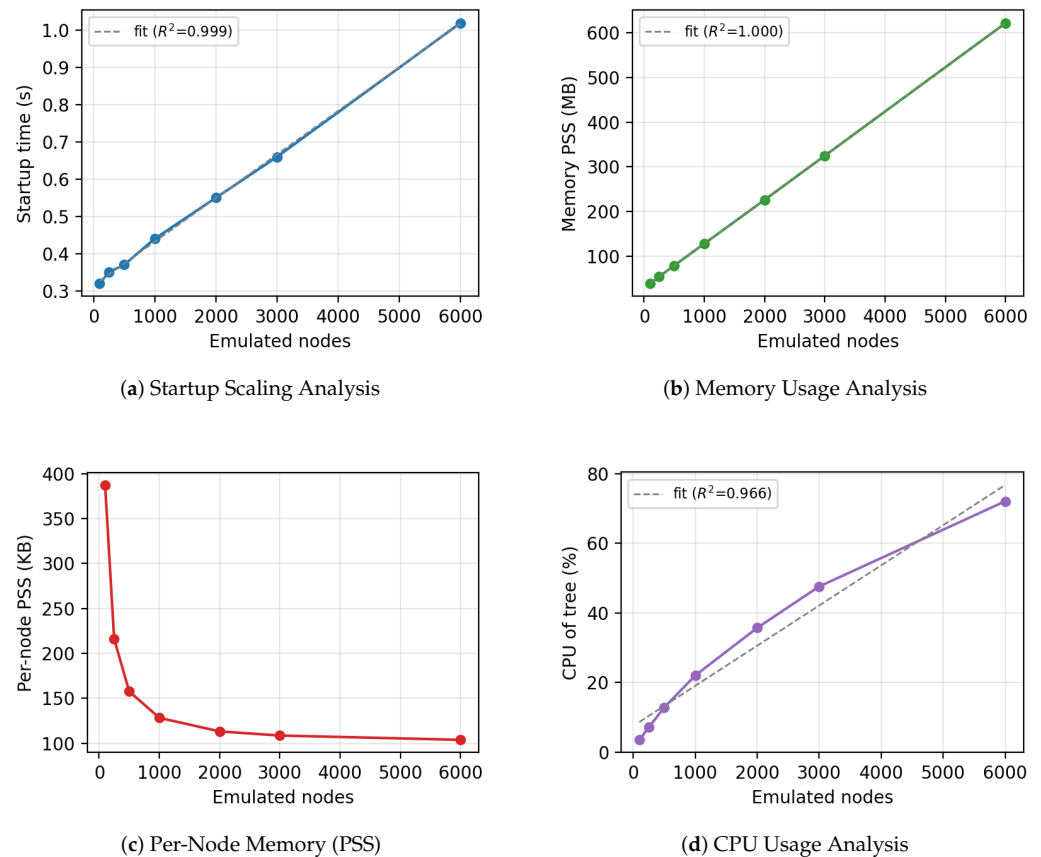


Figure 5. Process-based single-machine scalability on Workstation A. (a) Startup time and (b) memory (summed PSS) both scale linearly with node count ($R^2 = 0.9995$ and 1.000). (c) The per-node PSS falls as shared pages amortize across more processes, and (d) the CPU of the process tree rises with the aggregate 1 Hz offered load as the node count increases to 6000. Every node is an independent OS process.

Table 8. Resident memory of one custom-UDP node process (PSS, from smaps).

Region	PSS (KB)
Private C-runtime pages (libc/ld: GOT, TLS, writable data)	40
Anonymous data (sensor and sequence state, send buffer)	28
Binary text and data	20
Thread stack	16
Heap	4
Shared C-library text (proportional, amortized across nodes)	10
Single-process total	118
Marginal cost per node (Equation (10) slope)	99

6.2. Protocol Latency and Throughput at Scale

Section 6.1 measures the cost of instantiating nodes as processes, independent of any protocol. This experiment instead fixes the offered load at 10 Hz per node for 10 s on loopback and drives it through a full protocol stack: custom UDP, MQTT over an embedded Mosquitto 2.0 broker, and CoAP over aiocoap at 100, 250, and 500 nodes, with three runs per point. Delivery is the fraction of the fixed offered load recorded within the window; a value below 100% reflects throughput saturation rather than packet loss. Loopback removes network transit, thereby isolating the orchestrator and broker from

the wide-area effects described in Section 6.5. For each delivered message, we record the orchestrator’s client-side send-path latency, from issuing the send call to the protocol confirming the message has left the client: for CoAP in confirmable mode, this includes the request/acknowledgment round-trip; for MQTT at QoS 0, it is the publish enqueue onto the established TCP session. For custom UDP, it is the datagram write. The three quantities measure each protocol’s per-message client-side cost under its own reliability model, not a single common end-to-end delay. Table 9 reports the result.

Table 9. Per-protocol latency and delivery as the node count rises on the loopback interface (mean of three runs; MQTT at 500 nodes is a single run). Delivery is the fraction of the offered load (nodes $\times$ 10 Hz $\times$ 10 s) recorded at the receiver. P95 and P99 are per-message client-side send-path latency for delivered messages, defined per protocol in the text.

Protocol	Nodes	Delivery (%)	P95 (ms)	P99 (ms)
Custom UDP	100	100 [†]	0.038	0.050
Custom UDP	250	100 [†]	0.026	0.036
Custom UDP	500	Failed [‡]	–	–
MQTT	100	100	0.422	0.487
MQTT	250	100	0.341	0.392
MQTT	500	0.7	0.385	0.735
CoAP	100	100	0.898	0.997
CoAP	250	52.9	0.804	0.879
CoAP	500	25.8	0.839	1.039

[†] The custom UDP C client transmits slightly above the nominal rate, so its recorded count exceeds the offered load; we report this as full delivery. [‡] At 500 nodes, the passive-mode driver did not produce a receipt log in any of the three runs (one-process-per-node socket and PID pressure on the test host).

Two observations are particularly noteworthy. First, per-message P95 latency does not grow with node count for any protocol. It stays near 0.03 ms for custom UDP, between 0.34 and 0.42 ms for MQTT, and between 0.80 and 0.90 ms for CoAP across the whole range, so the round-robin orchestrator builds no queuing backlog per delivered message as concurrency rises. We emphasize that these three quantities are not a like-for-like “protocol overhead” comparison: each measures the client-side cost of a different reliability action (a datagram write, a QoS 0 enqueue onto an established TCP session, and a confirmable request/acknowledgment round-trip, respectively), as defined above. The ordering custom of UDP < MQTT < CoAP therefore reflects where each protocol places its reliability work, and its value here is as a per-protocol stability baseline across node counts rather than as a ranking. The ordering is statistically robust rather than measurement noise. On a matched 40-node loopback run that records every per-message send-path latency ($n = 5280$ per protocol), a two-sided Mann–Whitney U-test rejects the equality of the MQTT and CoAP distributions ($U = 1.65 \times 10^4$, $p < 0.001$, rank-biserial effect size of 0.999). The bootstrap 95% confidence intervals on the medians do not overlap (MQTT, 0.219–0.221 ms; CoAP, 0.614–0.617 ms). The separation is by construction of the two reliability models, namely the CoAP confirmable round-trip and the MQTT QoS 0 enqueue. The test confirms that this per-message cost difference is consistent across the full sample rather than an artifact of a few outliers. We use a nonparametric test because the latency samples are heavy-tailed and not normally distributed.

Second, the delivery column exposes a separate limit that is not visible in the latency figures, and it is the rationale for STGen’s process-based node model. The custom UDP client, i.e., one OS process per node, sustains the full offered load at every size. MQTT, whose paho clients multiplex over a shared thread pool, holds 100% delivery to 250 nodes and

collapses at 500 (341 of 50,000 messages delivered) because the Python global interpreter lock serializes the threaded publish path. The 500-node MQTT entry in Table 9 is a single completed run and is reported as a saturation indicator, not a performance estimate: in repeated attempts, the 500-client MQTT workload does not complete within the allotted run window, which is, itself, the saturation phenomenon under study, so no stable multi-run average exists at this operating point. CoAP, driven from a single `asyncio` event loop, saturates earlier (52.9% at 250, 25.8% at 500) because each confirmable exchange waits for its acknowledgment inside that loop. In every case, the bottleneck is the protocol client library operating at a fixed offered rate, not STGen's core, whose own latency (the custom UDP rows) remains flat. Processes cost more memory than threads or coroutines. Still, that cost is the 99 KB per node measured in Section 6.1, and in return, each node gets its own interpreter and scheduler context, which is what lets the instantiation sweep reach 6000 nodes without the client-library saturation seen here. A deployment requiring thousands of live broker or confirmable endpoints would distribute those clients across multiple hosts in the distributed model (Section 3.3) rather than collocate them on a single orchestrator.

6.3. Validation of Sensor Fidelity Against Real-World Data

To substantiate the high-fidelity claim quantitatively, we compare STGen-generated synthetic sensor streams against real-world readings from the Intel Berkeley Research Laboratory (IBRL) dataset [48], which contains readings collected over 37 days from 54 Mica2Dot mote sensors sampling temperature, relative humidity, ambient light intensity, and supply voltage at an average interval of approximately 31 s. After removing 93,879 malformed records and filtering 393,580 physically implausible outliers, the cleaned dataset contains 1,826,223 valid readings drawn from 54 independent sensor nodes. We generated 100,000 synthetic samples per modality using STGen's stochastic sensor engine and quantified statistical agreement using two complementary metrics: the two-sample Kolmogorov–Smirnov D-statistic (lower is better), which measures the maximum absolute difference between empirical CDFs, and the Jensen–Shannon Divergence (JSD; lower is better), which quantifies the information-theoretic distance between probability distributions on $[0, 1]$.

Table 10 summarizes the marginal statistics. Figure 6 plots histograms, the empirical CDF, and Q-Q alignment for all four modalities, so the visual validation covers humidity, light, and voltage, as well as temperature. The synthetic and real temperature distributions are very similar (KS $D = 0.071$ and JSD = 0.018), as shown in the figure. The difference between the sample means was just 0.03 °C. To quantify the statistical uncertainty of these metrics, we bootstrapped both statistics with 200 resampling replicates per modality (synthetic streams were resampled in full; real streams were subsampled to 200,000 readings per replicate). The percentile 95% confidence intervals are narrow for every modality: temperature, KS D [0.068, 0.076] and JSD [0.017, 0.019]; humidity, KS D [0.135, 0.141] and JSD [0.043, 0.045]; light, KS D [0.326, 0.334] and JSD [0.124, 0.128]; voltage, KS D [0.100, 0.105] and JSD [0.032, 0.038]. The reported point estimates are therefore stable to resampling, and the ordering of modality fidelity is not an artifact of sample noise. Voltage also matched closely (KS $D = 0.101$), owing to our slow-discharge model for Mica2Dot batteries. Humidity matched the real-world mean (39.53% vs. 39.54%) but exhibited a narrower spread ($\sigma = 4.28\%$ vs. 6.28%). This variance difference was expected. The calibration was performed based on the median behavior of the motes. Still, the physical deployment of the IBRL placed motes near drafty windows and HVAC (Heating, Ventilation, and Air Conditioning) vents, which caused a high standard deviation in mote behavior.

Table 10. Comparison of the marginal distributions of 1.82 million real Mica2Dot readings with the STGen synthetic data. We use the two-sample Kolmogorov–Smirnov statistic (KS D) and Jensen–Shannon Divergence (JSD) to measure distributional fidelity. The temperature modality showed the closest agreement between the real and synthetic data, with a mean difference of 0.03 °C.

Modality	Intel Lab		STGen		KS D	JSD
	μ	σ	μ	σ		
Temperature (°C)	22.43	4.19	22.40	3.37	0.071	0.018
Humidity (%)	39.54	6.28	39.53	4.28	0.138	0.044
Light (lux), default	409.3	541.4	515.4	384.5	0.330	0.125
Light (lux), GMM [†]	409.3	541.4	420.8	537.1	0.065	0.028
Voltage (V)	2.56	0.12	2.55	0.15	0.101	0.032

[†] Optional five-component Gaussian-mixture light profile, fitted on 27 motes and evaluated on the disjoint 27 (held-out); see Section 6.3. **Bold values indicate the improved distributional-fidelity metrics achieved by the optional Gaussian-mixture light profile.**

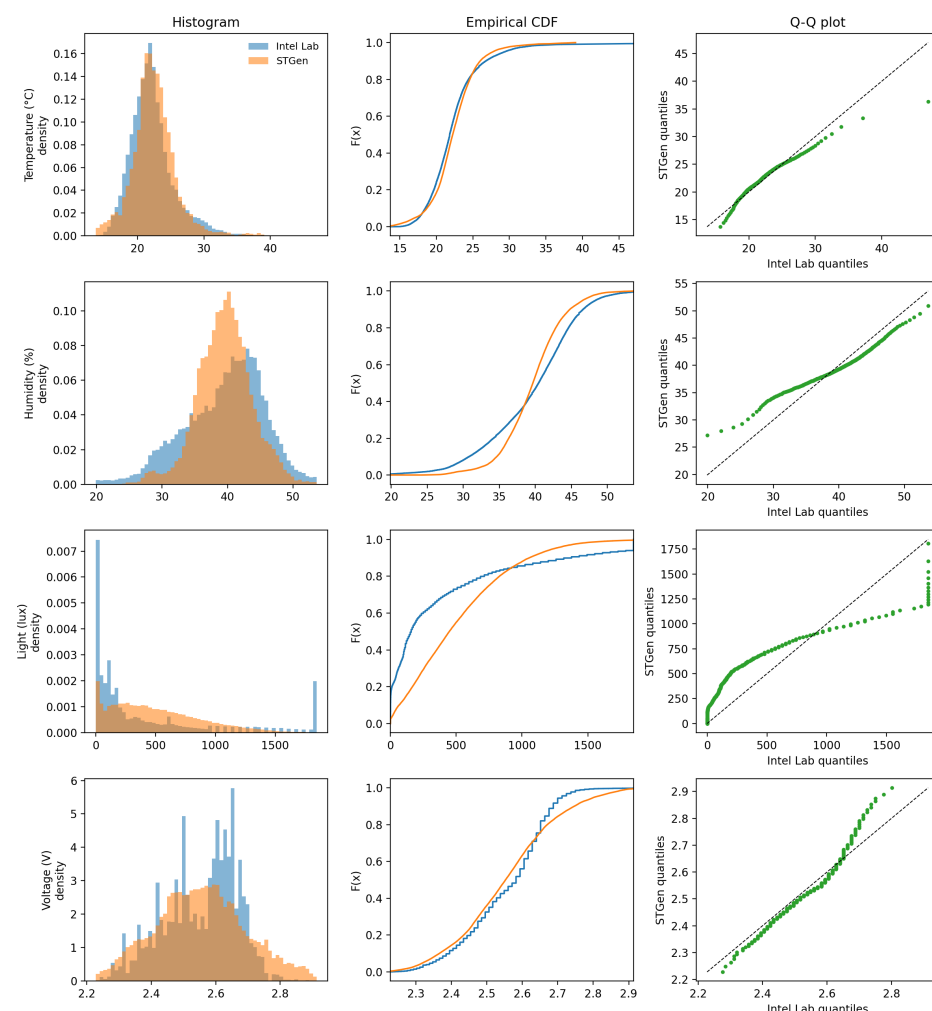


Figure 6. Distribution comparison for all four calibrated modalities (rows: temperature, humidity, light, and voltage) between the Intel Berkeley deployment (blue) and STGen synthetic streams (orange): overlaid histograms (**left**), empirical CDFs (**middle**), and Q-Q plots (**right**). The panels visualize the marginal-fidelity metrics of Table 10: temperature and voltage align closely; humidity matches in mean but with a narrower synthetic spread; and light shows the bimodal dark/bright structure that the default single-regime generator does not capture, which motivates the optional Gaussian-mixture profile evaluated in the text.

The light modality was the most challenging to model with our default single-regime generator ($KS\ D = 0.330$). The physical IBRL light data is strongly bimodal, capturing both dark corridors and bright workspaces. These two modes are difficult to reproduce with a single equation. To address this, we tested an optional five-component Gaussian-mixture profile within STGen. We fitted the mixture using the expectation-maximization algorithm on 50% of the motes and tested it on the other 50% of motes. All five illumination regimes (off, dim, ambient, lit, and saturated) could be recovered, and $KS\ D$ was reduced to 0.065 by this custom mixture model.

Table 11 shows the temporal autocorrelation (ACF) at each tested lag. The synthetic and real temperature ACF curves differ by less than 0.035 at all lags, indicating that the OU process maintains the long-range temporal memory of a real Mica2Dot stream. The temperature–humidity coupling, generated via Cholesky-decomposed noise, yields a Pearson correlation of $r = -0.403$, compared with the measured -0.426 (5.4% relative error). Synthetic voltage matches the real voltage at short lags but decays faster by $\ell = 50$ because the battery-discharge logic shortens the effective time constant; this drift is negligible for the minute-scale protocol evaluations that STGen targets but would matter for multi-day energy studies.

Table 11. Temporal autocorrelation function (ACF) of real and synthetic traces at different lags. The synthetic temperature ACF is extremely similar to the actual Intel deployment, and the difference is less than 0.035 at all time lags. Additionally, the framework reasonably reproduces the environmental cross-modality coupling; for example, the synthetic $r = -0.403$ was very close to the measured $r = -0.426$.

Modality	Intel Lab ACF					STGen ACF				
	$\ell=1$	$\ell=5$	$\ell=10$	$\ell=20$	$\ell=50$	$\ell=1$	$\ell=5$	$\ell=10$	$\ell=20$	$\ell=50$
Temperature	0.998	0.990	0.978	0.951	0.876	0.998	0.992	0.983	0.966	0.911
Humidity	1.000	0.998	0.995	0.986	0.954	0.994	0.974	0.949	0.900	0.760
Light	0.998	0.988	0.975	0.960	0.913	0.976	0.876	0.754	0.541	0.141
Voltage	0.998	0.998	0.997	0.996	0.992	0.995	0.974	0.947	0.896	0.737

6.4. Adversarial Scenario Generation and Validation

The fidelity result of Section 6.3 also fixes the abnormal case: an anomaly generated inside the testbed is a measurable departure from a model with a known statistical baseline, not an arbitrary numeric corruption. We use this property to produce labeled false-data-injection (FDI) datasets at the application layer, measured with the same Kolmogorov–Smirnov statistic that established fidelity.

The adversarial engine is deterministic and seeded; Algorithm 2 specifies how it injects the six attack families, each a transform of the true reading that crosses a named physical invariant (the absolute range, the operating range, the maximum plausible step, or the noise floor). It operates as a Tier-2 scenario setting that inverts the sensor profile of the selected nodes, with no change to the protocol or network tiers. Each injected record carries ground-truth labels (`is_attack`, `attack_type`, `true_value`, and `injected_value`), which support supervised training and evaluation of intrusion-detection methods. The fraction of malicious nodes, the attack family and its intensity, and the active window are set from the JSON configuration. We exercise the engine on seven sensor modalities: the four validated against the Intel Berkeley deployment in Section 6.3 (temperature, humidity, light, and voltage) and three further modalities (pressure, CO_2 , and sound) whose physical invariants are drawn from datasheet operating ranges rather than validated against that dataset. This yields 42 modality-by-family combinations through a single interface; the invariant table ($\mathcal{I}[\tau]$) provides the absolute range, operating range, noise floor, and maximum step for each modality.

Algorithm 2 Labeled Physically-Invariant Anomaly Injection

Require: True-reading stream (i, τ, x, t) ; invariant table $\mathcal{I}[\tau] = \{[lo, hi], \sigma_\tau, \Delta_\tau\}$ for each of the seven supported modalities τ

Require: Attack family A , malicious fraction f , intensity I , seed s , window $[t_0, t_1]$, per-message probability p

Ensure: labeled stream (x', ℓ) with ground-truth label ℓ

```

1: rng  $\leftarrow$  SEEDDRNG( $s$ ) ▷ reproducible node selection and noise
2:  $\mathcal{M} \leftarrow$  sample  $\lfloor fN \rfloor$  node indices with rng ▷ malicious set
3: initialize per-node state  $\{bias_i, stuck_i, buf_i, last_i\}$ 
4: for each message  $(i, \tau, x, t)$  in the stream do
5:   if  $i \in \mathcal{M}$  and  $t_0 \leq t \leq t_1$  and rng()  $< p$  then
6:      $x' \leftarrow$  INJECT( $A, x, \mathcal{I}[\tau], I, state_i$ )
7:      $\ell \leftarrow \langle \text{ATTACK}, A, x, x' \rangle$  ▷ ground-truth label
8:   else
9:      $x' \leftarrow x$ ;  $\ell \leftarrow \langle \text{NORMAL} \rangle$ 
10:  end if
11:   $state_i.last \leftarrow x$ ; emit  $(x', \ell)$ 
12: end for

13: function INJECT( $A, x, ([lo, hi], \sigma, \Delta), I, st$ )
14:   if  $A = \text{OUT-OF-RANGE}$  then
15:     return  $hi + I\sigma$  ▷ crosses absolute range
16:   else if  $A = \text{BIAS-DRIFT}$  then
17:      $st.bias \mathrel{+}= 0.1 I\sigma$ ; return  $x + st.bias$  ▷ leaves operating range
18:   else if  $A = \text{IMPOSSIBLE-JUMP}$  then
19:      $d \sim \{-1, +1\}$ ; return  $st.last + d(I+1)\Delta$  ▷ crosses step bound
20:   else if  $A = \text{VARIANCE-BURST}$  then
21:     return  $x + \mathcal{N}(0, (I\sigma)^2)$  ▷ exceeds noise floor
22:   else if  $A = \text{STUCK-AT}$  then
23:     if  $st.stuck = \emptyset$  then  $st.stuck \leftarrow x$ ; return  $st.stuck$ 
24:   else if  $A = \text{REPLAY}$  then
25:     append  $x$  to  $st.buf$ ; return random element of  $st.buf$  via rng
26:   end if
27: end function

```

To measure separability, we evaluate each labeled stream with a one-class detector whose normal model is the calibrated physical noise floor of the modality. The detector is a fixed rule with no trained parameters, so there is no train/test split and no risk of overfitting: the AUC measures how far an anomaly departs from the physical baseline, not the skill of a learned classifier. The detector score of a sample is the larger of a value residual (distance from the operating mean in noise-floor units) and a step residual (single-step change in units of the maximum plausible step). For every modality-by-family cell (792 injected samples each), we compute the physically invariant violation rate and the receiver-operating-characteristic area under the curve (AUC), with the latter serving as the rank statistic ($U/(n_+n_-)$), so it requires no separate classifier; precision, recall, and F_1 at a three-sigma operating point are recorded per cell in the released results. Table 12 aggregates the violation rate and AUC across the seven modalities, and Figure 7a gives the per-cell AUC for all 42 combinations.

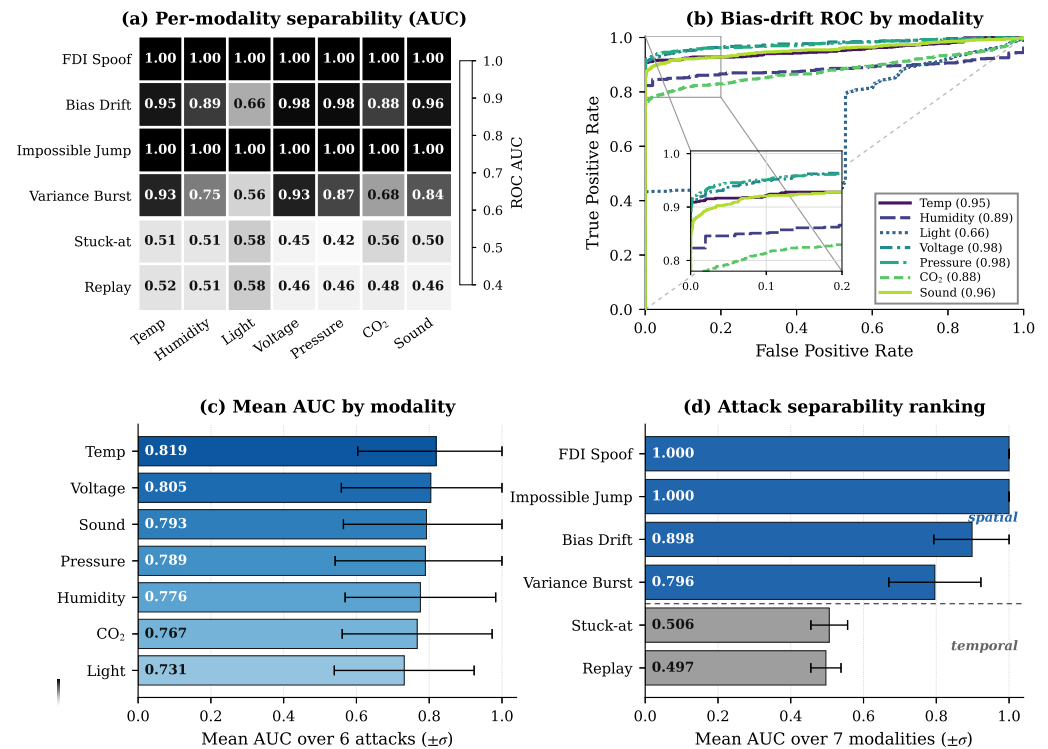


Figure 7. Adversarial anomaly validation over seven modalities and six families (42 cells; 792 injected samples each). The AUC is the separability of a fixed, untrained value/step detector (no train/test split). (a) Per-cell AUC; spatial families above the rule, temporal below. (b) Per-modality bias-drift ROC (the most modality-dependent family) with a low-FPR zoom inset; light (dotted) is the outlier. (c) Mean AUC per modality ($\pm 1\sigma$ over the six families). (d) Mean AUC per family ($\pm 1\sigma$ over the seven modalities); spatial versus temporal. The AUC 1.000 of out-of-range spoof and impossible jump follow from their deterministic invariant break, not from fitting.

Table 12. Adversarial anomaly validation aggregated over the seven modalities (temperature, humidity, light, voltage, pressure, CO₂, and sound; 792 injected samples per modality-by-family cell and 42 cells). Phys. viol. is the mean fraction of injected samples crossing an absolute-range, operating-range, or step bound; AUC is the one-class value/step detector's ROC area, reported as the mean $\pm$ standard deviation and the [min, max] across the seven modalities. The per-cell values appear in Figure 7a.

Attack Family	Phys. Viol. (%)	AUC (Mean $\pm$ σ)	AUC [Min, Max]
Out-of-range spoof	100.0	1.000 $\pm$ 0.000	[1.000, 1.000]
Impossible jump	99.3	1.000 $\pm$ 0.000	[1.000, 1.000]
Bias drift	79.2	0.898 $\pm$ 0.105	[0.657, 0.977]
Variance burst	66.6	0.796 $\pm$ 0.127	[0.564, 0.934]
Stuck-at	1.6	0.506 $\pm$ 0.050	[0.425, 0.583]
Replay	5.4	0.497 $\pm$ 0.042	[0.459, 0.584]

Three readings follow from Table 12 and Figure 7. First, the hard invariant breaks, out-of-range spoofing, and impossible jumps place essentially every injected sample outside the physical envelope and separate at AUC 1.000 on all seven modalities; for the impossible jump, the injected value distribution still overlaps the normal one (two-sample Kolmogorov–Smirnov $D = 0.524$ on temperature), so the separation comes from the step residual, not the value. Second, the stealthy families are modality-dependent: bias drift averages an AUC of 0.898, and variance burst is 0.796 across the seven modalities, with recall set by

how much of each injection stays inside the noise band (on temperature, bias-drift recall is 0.86, and variance-burst recall is 0.47 at three sigma, the latter of which is lower because part of each burst remains within the noise floor). Bias drift shows the widest spread across modalities (AUC of 0.657 to 0.977), so Figure 7b resolves it per modality rather than the families whose AUC is constant across modalities.

Third, the weakest modality is light (mean AUC of 0.731, bias-drift AUC of 0.657, and variance-burst AUC of 0.564). Its calibrated model is bimodal: a node alternates between a dark and a bright regime, so a slowly injected drift or a short variance burst under a value/step rule is not distinguishable from a legal regime change. The other six modalities lie in a narrow band (mean AUC of 0.767 to 0.819), so the engine and its measurement carry over from temperature, the modality analyzed in detail above, to the full set.

The validation also states that the value/step detector does not separate the attack and that the boundary is the same across all modalities. Stuck-at and replay average AUCs of 0.506 and 0.497 next to the no-skill diagonal, because they are temporal rather than value anomalies: a frozen or replayed reading is individually plausible. It crosses no value or step bound (mean physical-violation rates of 1.6% and 5.4%). The detector considered here is a measurement instrument for separability, not a proposed defense, so this is the expected result, and it marks the boundary of a value-domain test. The 42 labeled profiles serve as benchmarks for the temporal-variance or sequential detection methods that a value/step rule does not address.

We deliberately do not benchmark these anomalies against public IoT intrusion-detection datasets such as TON_IoT [49], Bot-IoT [50], or CICIoT2023 [51] because the two operate at different abstraction levels: those corpora label network-layer attack traffic (scanning, DDoS, or botnet command-and-control) captured as packet flows, whereas STGen's engine labels application-layer sensor-value falsification, whose realism is anchored in the physically calibrated baseline of Section 6.3 rather than in traffic signatures. A flow-level comparison would therefore not measure the property the engine provides: ground-truth-labeled departures from a statistically validated physical process. The two resources are complementary: network-layer datasets exercise flow-based detectors, and STGen's FDI profiles exercise value- and sequence-based detectors, with the added property that the normal class, itself, is validated against real deployment data.

6.5. Protocol Performance Benchmarking

We compared MQTT (TCP-based) [52] and CoAP (UDP-based) [53] under a single emulated congested link, holding everything except the transport stack fixed.

6.5.1. Experimental Design

We used the Smart Agriculture scenario (Figure 2) with 20 low-power sensor nodes, each generating periodic 112-byte packets (12-byte header, 100-byte payload). The congested profile applies a 150 ms base latency, 30 ms Gaussian jitter (50% within 10 ms), 5% uniform packet loss, and a 1000 kbps bandwidth cap, approximating a cellular backhaul or LoRaWAN link near its limits; all environmental parameters were held identical for both protocols. MQTT ran against an embedded Mosquitto 2.0 broker at QoS 0, so reliability rests solely on the TCP connection. In contrast, CoAP ran in confirmable (CON) mode with the RFC 7252 defaults (NSTART = 1, MAX_RETRANSMIT = 4, and ACK_TIMEOUT = 2 s), placing reliability at the application layer. MQTT QoS 1/2 and CoAP non-confirmable mode are out of scope (Section 8). Three 60-second runs per protocol were averaged.

Each value in Table 13 is a mean over three 60-second runs. At the scenario's 0.017 Hz rate, each run carries only about 20 messages (~60 pooled), so with CoAP's 694 ms spread, the mean-latency gap is not, by itself, statistically separable, and the CoAP P95 of

2574 ms should be read as evidence that the retransmission tail exists rather than as a precise quantile. What survives the dispersion is the loss rate (CoAP 10.4% versus MQTT 0%) and the P95 ordering, both corroborated by the 34 $rc=16$ reconnection events logged across the five distributed MQTT windows and confirmed at scale by the Mann–Whitney test of Section 6.2.

Table 13. Protocol latency analysis (smart agriculture scenario).

Protocol	Avg Latency	P50 Latency	P95 Latency	Std Dev	Loss (%)
CoAP (UDP)	528.95 ms	316.28 ms	2574.26 ms	694.01 ms	10.4
MQTT (TCP)	580.41 ms	481.65 ms	1270.18 ms	258.20 ms	0.0

Network condition: 150 ms delay, 5% loss, and 60 s duration. **Note:** Bold values indicate the best-performing metric in each respective column.

6.5.2. Performance Analysis

Figure 8 and Table 13 give the results. Three points stand out.

- **Median Latency:** CoAP's P50 of 316.28 ms sits within about 16 ms of the emulated link's minimum RTT ($2 \times 150 = 300$ ms), so a non-retransmitted confirmable exchange adds almost no processing delay. MQTT's 52.3% higher median (481.65 ms) traces to TCP session re-establishment. After a failed reconnect (logged as $rc=16$ at the broker), the client waits at least one full RTT before the next payload. This reconnection tax inflates the median independently of delivery.
- **Tail Latency and Retransmission Cost:** The protocols differ most at the tail. MQTT holds a P95 of 1270.18 ms ($\sigma = 258.20$ ms), while CoAP reaches 2574.26 ms ($\sigma = 694.01$ ms). The CoAP tail follows deterministically from RFC 7252's confirmable retransmission schedule (NSTART = 1, MAX_RETRANSMIT = 4, and ACK_TIMEOUT = 2 s with exponential backoff), which spreads one lost CON message across up to five transmit-and-listen cycles over a 62 s window; at 5% loss, the 20-node, 60 s workload generates enough exchanges entering this ladder to populate the observed tail. For battery-constrained nodes, the cost of the same ladder is up to four additional wake/transmit/listen duty cycles per lost message. STGen operates above OSI Layer 4 and does not convert this into a per-node joule figure, which would require the duty-cycle parameters of a specific radio; instead, it measures the latency consequence directly, as shown in Table 13. MQTT's zero loss avoids these cycles because TCP retransmits in-stack without an application-visible multi-second tail.
- **Observed Trade-offs, Not Deployment Mandates:** These results hold for the evaluated libraries and parameters (Eclipse Paho MQTT at QoS 0, aiocoap in CON mode) under a single congested profile and workload. The robust direction is that MQTT delivered every message with bounded variance. At the same time, CoAP incurred a 10.4% loss in exchange for a lower median because the two protocols place reliability at different layers. This ordering is regime-dependent: Section 6.8 shows it inverts under bursty wide-area loss, where MQTT's persistent session becomes the liability. The contribution is that STGen surfaces this regime dependence, not that one protocol is universally better; workloads requiring both low latency and high reliability motivate hybrids (e.g., MQTT-SN over DTLS or application-layer FEC) that the pluggable ProtocolInterface can host.

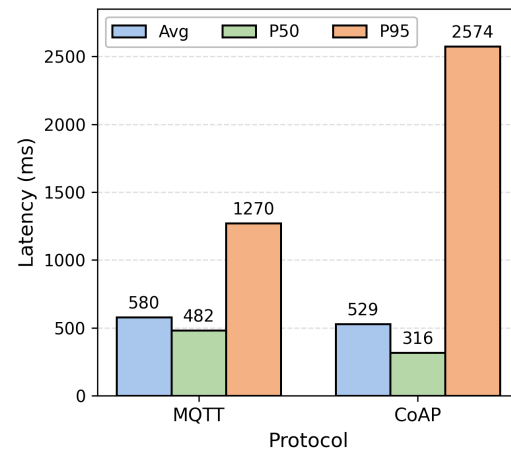


Figure 8. Protocol Latency Comparison: Median (P50) vs. Tail (P95) Latency under congested network conditions. CoAP exhibits pronounced tail-latency spikes and message loss compared to MQTT’s stable delivery behavior.

6.6. Memory Footprint Versus Virtualization-Based Testbeds

Figure 9 places STGen’s measured memory next to Gotham and GothX, the closest container- and VM-based IoT testbeds. From the GothX measurements, Gotham’s footprint follows $R = 37d + 470q$ MB for d Docker device nodes and q QEMU router VMs [9]: an emulated IoT device costs about 37 MB, so a 16 GB host is exhausted near 440 devices, and GothX reported 450 devices in 20.4 GB. STGen runs each node as an OS process at a marginal 99 KB (Equation 10), which is about 380 times less per node, and holds 6000 nodes in 622 MB on the same class of machine.

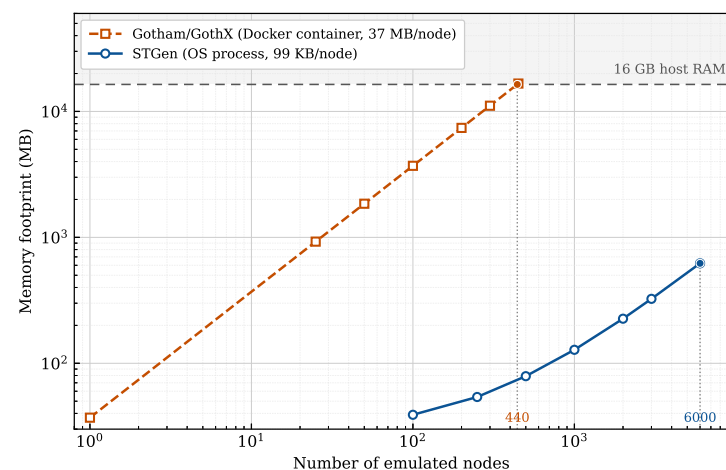


Figure 9. Memory footprint versus emulated node count (log–log). STGen’s measured process-tree PSS (blue) against the Gotham/GothX Docker per-node cost of 37 MB (red, from the GothX formula [9]); the shaded band marks memory beyond the 16 GB host limit. STGen holds 6000 nodes in 0.62 GB, whereas the container-based curve crosses 16 GB near 440 IoT devices.

This comparison is indicative rather than a controlled head-to-head benchmark: Gotham’s per-node figure is taken from the published GothX measurements, not re-executed on our hardware, and it covers only memory costs. Gotham buys full OS isolation per node (container or VM) for security dataset generation; STGen shares one kernel across processes for protocol evaluation. The per-node gap is the price of that isolation, and it sets the node ceiling on commodity hardware: a few hundred isolated containers against several thousand processes.

6.7. Distributed Network Configuration

To evaluate STGen under non-deterministic network conditions, distributed experiments were conducted across a private overlay mesh implemented with Tailscale, which is built on WireGuard [54]. The deployment consisted of three workstations at fixed locations in Dhaka, Bangladesh: Basabo (23.74° N, 90.43° E), South Shahjahanpur (23.72° N, 90.43° E), and Demra (23.68° N, 90.43° E), pairwise within 10 km of each other. The measured baseline round-trip time of 180–200 ms far exceeds what a sub-10 km direct fiber path would incur, indicating that at least some flows were carried over Tailscale’s DERP relays rather than peer-to-peer. Therefore, the absolute latency reflects overlay routing, not inter-site distance; what the setup provides is delivery over a live Internet overlay with real (measured, not synthesized) jitter, packet loss, and TCP reconnection behavior and no mobility, fading, or coverage model applied.

6.8. Cross-Protocol Distributed Performance Analysis

Table 14 summarizes the distributed runs. Over the wide-area overlay, CoAP and MQTT (QoS 0) diverge sharply in loss and tail latency, and the gap is driven by the transport layer rather than by the application workload, which was identical for both.

Experimental Fairness Conditions: Both protocols ran the same workload under the same conditions: MQTT with a 60 s keep-alive, QoS 0, and clean sessions against a Mosquitto 2.0 broker co-located with the STGen core on Workstation C (Demra) to avoid extra network hops; CoAP with the standard RFC 7252 back-off (NSTART = 1, MAX_RETRANSMIT = 4, ACK_TIMEOUT = 2 s). End-to-end latencies were measured with paired `time.perf_counter()` calls recorded by the orchestrator over five independent 60 s windows per protocol, from which the mean loss, mean latency, and P95 figures are computed.

- **Reliability Inversion and Its Failure Signature:** Under controlled emulation, CoAP lost 10.4% of messages, while MQTT dropped none; over the live overlay, the ordering inverts, with MQTT at 10.45% loss (up to 43.51% in individual runs) against CoAP’s 2.19%. NetEm models uniform, steady-state congestion, whereas live Internet loss is bursty. Under burst loss, a persistent TCP connection becomes a liability: a lost-segment burst stalls the stream, the keep-alive timer expires, and the whole MQTT session fails. The broker logs confirm the mechanism, recording 34 `rc = 16` (connection lost) events across the five MQTT windows, each a full TCP teardown followed by a reconnect-and-resubscribe cycle of 180–420 ms during which QoS 0 discards all in-flight messages. CoAP has no analogous cascade: each datagram is processed in isolation, so a drop costs one reading, and jitter that kills an MQTT session merely adds latency.
- **Latency Under Wide-Area Jitter:** CoAP’s mean end-to-end latency of 195.45 ms is within about 15 ms of the measured 180–200 ms baseline RTT. In comparison, MQTT’s 506.49 ms mean is inflated by the same reconnect cycles: each `rc = 16` event adds a TCP three-way handshake plus broker CONNACK round trip before the next delivery.
- **P95 Tail Divergence:** The P95 gap between CoAP (390.43 ms) and MQTT (1768.61 ms) represents the worst-case cost of TCP’s session model under head-of-line blocking [55]. MQTT’s tail exceeds common soft real-time IoT deadlines (roughly 1 s for building automation and 500 ms for industrial feedback). Therefore, for protocols intended for metropolitan use, STGen’s distributed mode gives a more realistic tail-latency reading than the emulated NetEm mode.

Table 14. Aggregated performance summary of MQTT and CoAP in distributed metropolitan nodes (A, B, and C).

Protocol	Avg. Loss (%)	Avg. Latency (ms)	P95 Latency (ms)	Resilience
MQTT (TCP)	10.45	506.49	1768.61	Low
CoAP (UDP)	2.19	195.45	390.43	High

7. Extensibility and Protocol Integration

STGen decouples the application layer and transport layer from the engine. The orchestrator runs independently of any specific communication protocol, framing format, or connection model. This architecture is based on the strategy behavioral pattern [56], which defines a family of interchangeable algorithms behind a common interface. In STGen, the interchangeable strategy involves the transport protocol, and the core is the client. To apply the Dependency Inversion Principle [57], the core depends only on an abstract `ProtocolInterface` (Table 15) rather than concrete implementations.

Table 15. Formal contract of the `ProtocolInterface` base class. Any protocol integrated into STGen must override all four abstract methods. The framework invokes them in the order shown; `send_data()` executes inside a tight generation loop between steps 2 and 4. Passive-mode protocols may delegate transport execution to external subprocesses inside `start_server()` and `start_clients()`, using `send_data()` solely to serialize the payload.

Order	Method	Signature	Responsibility
1	<code>start_server()</code>	<code>() → None</code>	Bind listener, broker, or sink node; must be non-blocking
2	<code>start_clients()</code>	<code>(n: int) → None</code>	Spawn <i>n</i> virtual sensor-client processes or threads
3	<code>send_data()</code>	<code>(cid: str, data: dict) → None</code>	Serialize STGen payload into protocol frame and transmit
4	<code>stop()</code>	<code>() → None</code>	Teardown all sockets, threads, and child processes

This yields a falsifiable claim about extensibility: any protocol whose communication life cycle maps to the `(start_server, start_clients, send_data, stop)` tuple integrates natively. This condition also defines the framework's logical boundary: it accommodates connection-oriented, datagram, request–response, and publish–subscribe transports but explicitly excludes physical-layer mechanisms. The claim is validated here for three transports (MQTT, CoAP, and a custom UDP protocol); the integration of additional paradigms, such as AMQP, DDS, and QUIC, remains for future work (Section 8).

7.1. Integration Workflow

Protocol integration strictly requires implementation of the `ProtocolInterface` defined in Table 15. The framework enforces a hard execution sequence. First, the orchestrator boots the server, spawns *n* clients, and executes a tight data-generation loop; finally, the system is torn down once the data-generation loop completes. We verified three structurally distinct families: connection-oriented MQTT via `paho`, connectionless CoAP via `aiocoap`, and a custom pub–sub protocol over raw UDP sockets to prove that this abstraction holds across distinct networking models.

STGen currently provides two integration modes. In active mode, the engine directly orchestrates protocol execution. In passive mode, external client and server binaries run independently while STGen passively extracts timing and loss metrics without interfering with the transport stack.

7.2. Custom Protocol Integration and Scenario Compatibility

To validate this extensibility, we integrated a custom UDP protocol (`CustomUDP`) by subclassing the interface in passive mode. Here, the `start_server()` and `start_clients()`

methods simply spawn external C binaries as background subprocesses. The `send_data()` method serializes the synthetic physics data into the custom UDP frame, and `stop()` issues POSIX termination signals to clean up the environment.

Crucially, once developers write a protocol adapter, it automatically inherits full scenario compatibility. A researcher can transition a Smart Agriculture workload, from testing MQTT to testing the custom UDP protocol, by altering a single JSON field (`"protocol": "mqtt" → "protocol": "custom_udp"`) in the Tier-2 configuration. This structural boundary ensures rapid, cross-domain protocol validation without forcing researchers to rebuild the underlying sensor topologies.

8. Limitations and Future Directions

All emulation frameworks operate within specific boundaries. This section defines the exact conditions under which STGen's measurements hold, identifies the limits of their generalizability, and states the directions in which those limits will be addressed. In summary, the current limitations are that network emulation is restricted to Linux hosts because it relies on the `tc/NetEm` subsystem; protocol validation covers three transports (MQTT, CoAP, and a custom UDP protocol); sensor calibration uses a single benchmark dataset (Intel Berkeley, indoor); the distributed evaluation spans one metropolitan overlay of three hosts; and all reported measurements were taken on a single operating-system family, so the cross-platform reproducibility of timing-sensitive results has not yet been established. The validity analysis is divided into discussions of internal and external validity.

8.1. Internal Validity

Internal validity evaluates the specific factors within our experimental design, calibration choices, and the execution environment, which can directly affect the consistency and precision of STGen's measurements. Specifically, we identify three primary threats to internal validity:

1. **Process-Based Emulation vs. Physical Hardware Fidelity:** STGen instantiates sensor nodes as OS processes; hence, it does not simulate physical-layer mechanics and ignores MAC-layer collisions and complex battery-discharge dynamics. Consequently, STGen cannot replace hardware-in-the-loop testing for lower-level mechanisms like IEEE 802.15.4 CSMA/CA or LoRa chirp modulation. Researchers developing strict duty-cycled MAC protocols should treat STGen data as an upper-bound performance baseline and require physical hardware for final validation. For the same reason, STGen is not a replacement for packet-level simulators such as ns-3, OMNeT++, or Cooja.
2. **Sensor-Model Calibration Scope:** We calibrated the Ornstein–Uhlenbeck temperature parameters ($\theta = 0.0755$ and $\sigma = 1.313$) and the temperature–humidity coupling ($\rho = -0.38$) using the IBRL indoor dataset [48]. This 37-day record is from a climate-controlled office with a median temperature of 22.4 °C and a thermal relaxation time of 4 hours. Manual recalibration of θ , μ , and σ is needed for deployments in radically different regimes, such as outdoor agriculture with large diurnal temperature variations, cold-chain logistics, or industrial furnaces. These constants can be changed directly in the JSON configuration, but the default parameters are carefully optimized for typical indoor WSNs.
3. **Hardware Dependency of Scalability Measurement:** The 6000-node scalability metrics (Table 7) were obtained on one machine, Workstation A, so the fitted startup and memory coefficients of Equations (9) and (10) reflect that specific CPU and OS scheduler and will shift on NUMA systems or older I/O subsystems. The near-unity

R^2 values indicate structural linearity, but external users need to execute the baseline scalability script on their own machines to predict maximum node density.

8.2. External Validity

The following points outline the environmental, architectural, and configuration factors that may limit the generalizability of STGen's findings to broader production environments:

1. **Network Emulation vs. Production Network Dynamics:** The protocol benchmarks are based on Linux NetEm at IP Layer 3 (Section 6.5) for the localized (single-machine) experiments. This engine introduces independent Bernoulli packet loss and Gaussian jitter without considering complex packet-loss phenomena such as correlated burst (Gilbert–Elliott) drops, TCP slow-start dynamics, or QUIC path migration. To compensate, the distributed experiments (Section 6) transmit traffic over a real wide-area Tailscale overlay. This reflects the asymmetry of paths and the TCP reconnection penalty in the real world, but the results come with two important caveats. The first is that the 180–200 ms RTT on a short geographic baseline is consistent with DERP relay routing rather than real-world distance. Second, testing was limited to a single metropolitan area. Transcontinental links will have totally different jitter profiles. Finally, the absolute millisecond figures do not transfer across networks, but the mechanistic conclusions (such as CoAP's tail latency increasing) do.
2. **Protocol Coverage and QoS Tier Scope:** This study isolates specific protocol configurations: MQTT in QoS 0 and CoAP in the default confirmable (CON) mode; MQTT QoS 1/2, CoAP over DTLS, and Observe subscriptions [58] were not evaluated. Those configurations add broker-side state tracking, per-session keys, or in-flight message stores, which alter throughput under loss and raise the per-node memory above the figures of Section 6.1, so the 99 KB per-node cost is a floor that must be re-measured for each heavier protocol, and engineers building secure or state-heavy infrastructure must validate at the appropriate QoS tier before deploying.
3. **Scenario Generalizability:** The emulated tests were based exclusively on the smart agriculture profile (sparse traffic, 20 nodes, and sub-1 Hz sampling). Pushing the testbed into dense urban sensing (thousands of nodes at high frequencies) or industrial robotics (sub-100 ms deadlines) could easily trigger Python GIL contention or kernel scheduling bottlenecks that a 20-node cluster hides. We tested loopback P95 stability up to 6000 nodes (Section 6.1) and mapped library saturation points for MQTT/CoAP (Section 6.2), but loopback isolates local CPU queuing. It completely ignores network-stack congestion. Measurement of true end-to-end latency across hundreds of heavily loaded, geographically separated nodes requires further investigation.
4. **Distributed Orchestration Scale:** The distributed evaluation involves three hosts reporting to a single central sink. This validates the deployment model but does not measure how orchestration overhead, clock-synchronization error, or the load on the central STGen Core grows with the number of participating hosts. Therefore, the single-machine scalability results (Section 6.1) cannot be assumed to transfer to large multi-host deployments, where sink-side aggregation may become the bottleneck before per-host resources do. To substantiate large-scale distributed capabilities, future evaluations will test the coordination framework across multi-node topologies.
5. **Cross-Platform Reproducibility:** All measurements reported in this article were performed on Linux hosts. Operating-system scheduling, process-creation cost, and network-stack behavior all influence timing-sensitive metrics, so the reported coefficients (startup slope, per-node memory, and latency percentiles) are Linux-specific. The experimental scripts are published so the sweeps can be re-executed elsewhere,

but reproducibility across Windows, macOS, and BSD execution environments has not been demonstrated.

8.3. Future Directions

Figure 10 outlines planned future work aimed at developing protocol-abstracting middleware for IoT devices. This middleware is intended to bridge heterogeneous IoT devices and real-world applications by abstracting the complexities of the underlying protocols. Consistent with the layer scope defined in Section 1, the ProtocolInterface of Table 15 spans both the application and the transport layers and imposes no assumption on whether the underlying transport is connection-oriented or datagram-based. Consequently, the roadmap includes integration of MQTT [59] and CoAP [60] alongside transport-layer protocols such as QUIC.

The planned extensions are described as follows.

- **Broader protocol and dataset coverage:** The extensibility claim of Section 7 is currently validated on three transports. The planned work integrates structurally different communication paradigms, including AMQP, DDS, and QUIC-based messaging, through the same ProtocolInterface. It also extends sensor calibration beyond the Intel Berkeley indoor dataset to outdoor and industrial deployment records so that the fidelity methodology of Section 6.3 is demonstrated in more than one regime.
- **Broader distributed validation:** The wide-area study used a single Tailscale overlay across one metropolitan area, and the 180–200 ms baseline indicates relayed paths rather than direct ones. Future runs will add direct-routed overlays and transcontinental sites so that overlay-routing effects can be separated from geographic distance and the loss and tail-latency findings can be confirmed across multiple network regimes.
- **Cross-platform network emulation:** STGen currently uses the Linux tc (Traffic Control) subsystem for network emulation. As a result, capabilities such as latency injection and bandwidth throttling are currently limited to Linux-based environments. Future work will investigate automated integration with WSL2 to ensure consistent behavior in Windows and macOS.
- **No-code experimental design:** A centralized Web-based user interface will be re-integrated to provide a no-code environment for experimental design. This interface will provide deep-link integration with the ELK stack, enabling real-time visual monitoring of P95 latency trends and overall system health across thousands of emulated nodes.
- **Adaptive protocol optimization:** Moreover, in the proposed middleware, edge intelligence will be integrated using Reinforcement Learning (RL)-based algorithms [61] to support dynamic protocol switching during runtime. This will enable runtime optimization for objectives such as energy and latency for the IoT devices.

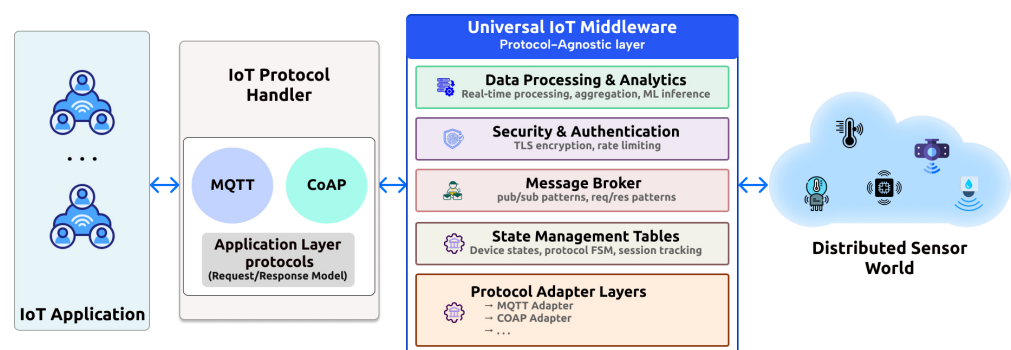


Figure 10. Universal IoT middleware architecture: foundation and directions for future work.

9. Conclusions

This paper introduces STGen, a sensor traffic generator for experimentation with transport-layer-and-above IoT protocols that emulates wireless sensor networks (WSNs) on commodity hardware. STGen targets three costs of physical-testbed deployment: hardware expense, resource limits, and limited scalability. It adopts a process-based design rather than heavy virtualization to address these challenges. Its three-tier architecture separates protocol, scenario, and network concerns, reducing the configuration effort from a multiplicative $N \times M \times L$ problem to an additive $N + M + L$ problem.

A central contribution of this work is the quantitative validation of STGen's stochastic sensor models against 1,826,223 real-world readings from the Intel Berkeley Research Laboratory's 54-node Mica2Dot deployment. For temperature, the synthetic and measured distributions agree with KS $D = 0.071$ and JSD = 0.018, with sample means within 0.12%. The Ornstein–Uhlenbeck process holds the autocorrelation within 0.035 of the measured curve across all tested lags. The temperature–humidity cross-correlation is $r = -0.403$ against the measured -0.426 , corresponding to a 5.4% relative error, which shows that the Cholesky-decomposed noise model carries the evaporative-cooling coupling between the two channels. These results establish that STGen's streams match both the marginal and the temporal statistics of real sensor data.

STGen supports multiple interaction interfaces, including a command-line interface (CLI), a Web-based dashboard, and RESTful API endpoints that comply with the OpenAPI specification. These interfaces support workflows from automating test environments and measuring protocol performance under load to generating traffic for network-security research. Container and VM-based testbeds reach a few hundred nodes on commodity hardware because each emulated IoT device carries about 37 MB of container overhead and each router VM carries about 470 MB [9]. STGen runs 6000 emulated nodes on a mid-range workstation, where each node is implemented as an independent OS process. The system starts in 1.02 s and uses 0.62 GB of memory (99 KB per node), which is more than two orders of magnitude less memory per node. Because STGen models only OSI Layer 4 and higher, these figures are an upper bound on protocol-layer performance: a physical MAC layer under localized spectrum congestion adds loss and delay that STGen does not reproduce, so the results read as a best-case baseline for the transport and application protocols under test. Within that scope, STGen exposed deployment failures that simplified simulators miss, including MQTT reconnect storms under wide-area jitter and CoAP retransmission tail latency.

The current validation scope bounds the conclusions drawn here: three communication protocols (MQTT, CoAP, and a custom UDP protocol), one calibration dataset (the indoor Intel Berkeley deployment), Linux execution environments, and a three-host metropolitan distributed setup. STGen is therefore presented as a validated methodology and working platform within that scope, not as a fully general evaluation framework. Future work will broaden this scope by incorporating additional protocol paradigms (e.g., AMQP, DDS, and QUIC), further calibration datasets, cross-platform execution, and larger distributed deployments, as outlined in Section 8.

Author Contributions: Conceptualization: H.M.A.I. and M.M.R.M.; methodology: H.M.A.I., M.M.R.M. and S.M.N.S.; software: M.M.R.M., S.M.N.S., P.A. and M.M.; validation: M.M.R.M., S.M.N.S., P.A. and M.M.; formal analysis: H.M.A.I., M.M.R.M. and M.G.; Writing: M.M.R.M., H.M.A.I. and M.G.; Review and editing: H.M.A.I., M.M.R.M., M.G., N.R.A., S.M.N.S., P.A. and M.M.; visualization: M.M.R.M., S.M.N.S., P.A. and N.R.A.; supervision: H.M.A.I., M.G. and R.I.; project administration: H.M.A.I. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The STGen source code, the protocol adapters, the three-tier configuration files, and the scripts that reproduce every experiment and figure in this article (the scalability sweep, the sensor-fidelity validation against the Intel Berkeley dataset, the protocol benchmarking, and the distributed evaluation) are available in a public repository at <https://github.com/DhmaiNetRG/STGen> (accessed on 28 July 2026). The Intel Berkeley Research Laboratory dataset used for fidelity validation is available at <https://db.csail.mit.edu/labdata/labdata.html> (accessed on 28 July 2026) [48].

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Command-Line Reference

Every experiment in this article was launched through a single headless entry point, `python -m stgen.main`; the REST API and the Web dashboard call the same path internally, so a run defined in the GUI reproduces what runs from the shell. Table A1 lists the orchestrator arguments. Appendix B maps each reported table and figure to the command that produced it, and Appendix C gives the code needed to register a new protocol.

Table A1. Arguments accepted by `python -m stgen.main`.

Argument	Description
<code>config</code>	Positional. Path to a JSON experiment file; when given, it overrides the individual flags below.
<code>-scenario name</code>	Scenario-tier preset, e.g., <code>smart_agriculture</code> , <code>smart_home</code> .
<code>-protocol name</code>	Protocol under test: <code>mqtt</code> , <code>coap</code> , or <code>custom_udp</code> .
<code>-compare list</code>	Comma-separated protocols to benchmark in a single run.
<code>-num-clients N</code>	Number of concurrent emulated sensor nodes.
<code>-duration S</code>	Run length in seconds.
<code>-inject-failures r</code>	Stochastic link-failure rate, $r \in [0.0, 1.0]$.
<code>-validate</code>	Run post-run integrity checks on the recorded telemetry.
<code>-list-scenarios</code>	Print the registered scenarios and exit.
<code>-list-protocols</code>	Print the registered protocols and exit.
<code>-debug</code>	Enable verbose logging.

```
# discover what is registered
python -m stgen.main --list-protocols
python -m stgen.main --list-scenarios

# run one protocol under a scenario
python -m stgen.main --scenario smart_agriculture --protocol coap \
    --num-clients 20 --duration 60

# benchmark several protocols in a single pass, then check integrity
python -m stgen.main --scenario smart_agriculture \
    --compare mqtt,coap,custom_udp --num-clients 20 --duration 60 --validate
```

Appendix B. Reproducing the Reported Results

Each table is produced by a named script that writes a result file under `results/`; the repository's `README.md` records the output file and expected runtime for each. Table A2 gives the commands. The scalability sweep additionally requires the custom-UDP node binaries (`make -C protocols/custom_udp`), and the emulated protocol comparison requires

root access because it applies the congested profile through the Linux tc/NetEm subsystem. The distributed evaluation needs three networked hosts joined by a Tailscale overlay and is launched with the commands that follow.

Table A2. Commands that regenerate each reported artifact.

Artifact	Script	Command
Table 7	run_scalability_test.py	python3 run_scalability_test.py
Table 9	run_protocol_scalability.py	python3 run_protocol_scalability.py
Section 6.2 (significance)	run_significance_test.py	python3 run_significance_test.py
Tables 10 and 11	run_fidelity_analysis.py	python3 run_fidelity_analysis.py
Fidelity CIs, Figure 6	run_fidelity_bootstrap.py	python3 run_fidelity_bootstrap.py
Regression CIs (Section 6.1)	run_regression_diagnostics.py	python3 run_regression_diagnostics.py
OU + light-GMM calibration (Section 6.3)	run_sensor_calibration.py	python3 run_sensor_calibration.py
Table 12	run_anomaly_validation.py	python3 run_anomaly_validation.py
Table 13	run_protocol_comparison.py	sudo python3 run_protocol_comparison.py
Table 14	distributed/	three hosts; see the commands below

The distributed run uses one core (sink) host and two sensor hosts on a shared overlay. On the core host (Workstation C):

```
python3 distributed/core_node.py --protocol mqtt --bind-ip 0.0.0.0 --duration 60
```

On each sensor host (Workstations A and B):

```
python3 distributed/sensor_node.py --core-ip <CORE_OVERLAY_IP> --core-port 5000 \
--node-id W1 --sensors 100 --protocol mqtt --duration 60
```

After the run, the per-host logs are merged into the aggregate of Table 14:

```
python3 distributed/aggregate_results.py
```

Appendix C. Integrating a New Protocol

A protocol is added by subclassing `ProtocolInterface` and overriding the four methods of Table 15; the orchestrator then selects it by name through `-protocol`, with no change to the core (Section 7). The skeleton below is the complete contract: `start_server` binds the listener, broker, or sink; `start_clients` creates the sender endpoints (as OS processes, threads, or async clients, depending on the transport needs); `send_data` serializes and transmits one payload and returns a `(delivered, timestamp)` pair, where `timestamp` is a `time.perf_counter()` reading taken once the transport has accepted the message, from which the orchestrator derives the per-message latency and increments its delivered count; and `stop` tears the run down. Because the timestamp is taken at the point that each transport considers the send complete, the recorded quantity is the client-side send-path cost under that protocol's own reliability model (Section 6.2).

```

from typing import Dict, Tuple
from stgen.protocol_interface import ProtocolInterface

class MyProtocol(ProtocolInterface):
    def start_server(self) -> None:
        # Bind the listener, broker, or sink. Must return promptly.
        ...

    def start_clients(self, num: int) -> None:
        # Create 'num' sender endpoints (process, thread, or async client).
        ...

    def send_data(self, client_id: str, data: Dict) -> Tuple[bool, float]:
        # Serialize one payload and transmit it. Return (delivered, timestamp),
        # where timestamp = time.perf_counter() taken once the transport has
        # accepted the message; the orchestrator turns it into a latency.
        return delivered, time.perf_counter()

    def stop(self) -> None:
        # Tear down sockets, threads, and child processes.
        ...

```

Once the class is on the import path, the same scenario and network-profile tiers apply unchanged. Hence, the protocol is immediately testable across all scenarios and impairment presets reported in Section 4.

References

1. Yick, J.; Mukherjee, B.; Ghosal, D. Wireless sensor network survey. *Comput. Netw.* **2008**, *52*, 2292–2330. <https://doi.org/10.1016/j.comnet.2008.04.002>.
2. Matin, M.A.; Islam, M. Overview of wireless sensor network. In *Wireless Sensor Networks-Technology and Protocols*; IntechOpen: London, UK, 2012.
3. Ramanathan, R.; Redi, J. A brief overview of ad hoc networks: Challenges and directions. *IEEE Commun. Mag.* **2002**, *40*, 20–22.
4. Sudevalayam, S.; Kulkarni, P. Energy Harvesting Sensor Nodes: Survey and Implications. *IEEE Commun. Surv. Tutor.* **2011**, *13*, 443–461. <https://doi.org/10.1109/SURV.2011.060710.00094>.
5. Saginbekov, S.; Shakenov, C. Testing wireless sensor networks with hybrid simulators. *arXiv* **2016**, arXiv:1602.01567.
6. Serôdio, C.; Cunha, J.; Candela, G.; Rodriguez, S.; Sousa, X.R.; Branco, F. The 6G Ecosystem as Support for IoE and Private Networks: Vision, Requirements, and Challenges. *Future Internet* **2023**, *15*, 348. <https://doi.org/10.3390/fi15110348>.
7. Swann, J.; Smith, L.; Jones, A. Tools for Network Traffic Generation—A Quantitative Comparison. *J. Netw. Syst. Manag.* **2021**, *29*, 1–19.
8. Sáez-de Cámara, X.; Flores, J.L.; Arellano, C.; Urbiet, A.; Zurutuza, U. Gotham testbed: A reproducible IoT testbed for security experiments and dataset generation. *IEEE Trans. Dependable Secur. Comput.* **2023**, *21*, 186–203.
9. Poisson, M.; Carnier, R.; Fukuda, K. GothX: A generator of customizable, legitimate and malicious IoT network traffic. In Proceedings of the 17th Cyber Security Experimentation and Test Workshop, Philadelphia, PA, USA, 13 August 2024; pp. 65–73.
10. Ghazanfar, S.; Hussain, F.; Rehman, A.U.; Fayyaz, U.U.; Shahzad, F.; Shah, G.A. IoT-flock: An open-source framework for IoT traffic generation. In Proceedings of the 2020 International Conference on Emerging Trends in Smart Technologies (ICETST), Karachi, Pakistan, 26–27 March 2020; pp. 1–6.
11. Bures, M.; Ahmed, B.S.; Rechtberger, V.; Klima, M.; Trnka, M.; Jaros, M.; Bellekens, X.; Almog, D.; Herout, P. Patriot: IoT automated interoperability and integration testing framework. In Proceedings of the 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), Virtual, 12–16 April 2021; pp. 454–459.
12. Li, R.; Li, Q.; Zou, Q.; Zhao, D.; Zeng, X.; Huang, Y.; Jiang, Y.; Lyu, F.; Ormazabal, G.; Singh, A.; et al. IoTGemini: Modeling IoT network behaviors for synthetic traffic generation. *IEEE Trans. Mob. Comput.* **2024**, *23*, 13240–13257. <https://doi.org/10.1109/TMC.2024.3426600>.
13. Sundresh, S.; Kim, W.; Agha, G. SENS: A sensor, environment and network simulator. In Proceedings of the Annual Simulation Symposium, Arlington, VA, USA, 18–22 April 2004; pp. 221–228. <https://doi.org/10.1109/simsym.2004.1299486>.
14. Varga, A.; Hornig, R. An overview of the OMNeT++ simulation environment. In Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops, Marseille, France, 3–7 March 2008; pp. 1–10.

15. Riley, G.F.; Henderson, T.R. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 15–34.
16. Velinov, A.; Mileva, A. Running and testing applications for Contiki OS using Cooja simulator. In Proceedings of the VII International Conference on Information Technology and Development of Education (ITRO), Zrenjanin, Serbia, 10 June 2016; pp. 283–289.
17. Postel, J. RFC 768; User Datagram Protocol; RFC, Ed.; RFC headquarters: Marina del Rey, CA, USA, 1980.
18. Fotiou, N.; Trossen, D.; Polyzos, G.C. Illustrating a publish-subscribe internet architecture. *Telecommun. Syst.* **2012**, *51*, 233–245.
19. Sachdeva, G.S. *Practical ELK Stack*; Apress: New York, NY, USA, 2017.
20. Adjih, C.; Baccelli, E.; Fleury, E.; Harter, G.; Mitton, N.; Noel, T.; Pissard-Gibollet, R.; Saint-Marcel, F.; Schreiner, G.; Vandaele, J.; et al. FIT IoT-LAB: A large scale open experimental IoT testbed. In Proceedings of the 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), Milan, Italy, 14–16 December 2015; pp. 459–464.
21. Sanchez, L.; Muñoz, L.; Galache, J.A.; Sotres, P.; Santana, J.R.; Gutierrez, V.; Ramdhany, R.; Gluhak, A.; Krco, S.; Theodoridis, E.; et al. SmartSantander: IoT experimentation over a smart city testbed. *Comput. Netw.* **2014**, *61*, 217–238.
22. Sundani, H.; Li, H.; Devabhaktuni, V.; Alam, M.; Bhattacharya, P. Wireless sensor network simulators a survey and comparisons. *Int. J. Comput. Netw.* **2011**, *2*, 249–265.
23. Kröllner, A.; Pfisterer, D.; Buschmann, C.; Fekete, S.P.; Fischer, S. Shawn: A new approach to simulating wireless sensor networks. *arXiv* **2005**, arXiv:cs/0502003.
24. Simon, G.; Volgyesi, P.; Maróti, M.; Ledeczi, A. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. In Proceedings of the IEEE Aerospace Conference, Big Sky, MT, USA, 8–15 March 2003; Volume 3, pp. 31339–31346.
25. Wightman, P.M.; Labrador, M.A. Atarraya: A simulation tool to teach and research topology control algorithms for wireless sensor networks. In Proceedings of the 2nd International Conference on Simulation Tools and Techniques, Rome, Italy, 2–6 March 2009; pp. 1–10.
26. Bounceur, A.; Clavier, L.; Combeau, P.; Marc, O.; Vauzelle, R.; Masserann, A.; Soler, J.; Euler, R.; Alwajeel, T.; Devendra, V.; et al. CupCarbon: A new platform for the design, simulation and 2D/3D visualization of radio propagation and interferences in IoT networks. In Proceedings of the 2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 12–15 January 2018; pp. 1–4.
27. Levis, P.; Lee, N.; Welsh, M.; Culler, D. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, Los Angeles, CA, USA, 5–7 November 2003*; SenSys '03; Association for Computing Machinery: New York, NY, USA, 2003; pp. 126–137. <https://doi.org/10.1145/958491.958506>.
28. Jha, D.N.; Alwasel, K.; Alshoshan, A.; Huang, X.; Naha, R.K.; Battula, S.K.; Garg, S.; Puthal, D.; James, P.; Zomaya, A.; et al. IoTsim-Edge: A simulation framework for modeling the behavior of Internet of Things and edge computing environments. *Soft. Pract. Exp.* **2020**, *50*, 844–867.
29. Nikolaidis, F.; Marazakis, M.; Bilas, A. IOTier: A Virtual Testbed to evaluate systems for IoT environments. In Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Melbourne, Australia, 10–13 May 2021; pp. 676–683.
30. Bendaouch, F.; Zaydi, H.; Merzouk, S.; Assoul, S. Benchmarking IoT Simulation Frameworks for Edge-Fog-Cloud Architectures: A Comparative and Experimental Study. *Future Internet* **2025**, *17*, 382.
31. Gil, S.; Mikkelsen, P.H.; Gomes, C.; Larsen, P.G. Survey on open-source digital twin frameworks—A case study approach. *Soft. Pract. Exp.* **2024**, *54*, 929–960.
32. Paul, T.C.; Lertpongrijorn, P.; Nguyen, H.D.; Salehi, M.A. Benchmarking Message Brokers for IoT Edge Computing: A Comprehensive Performance Study. *arXiv* **2026**, arXiv:2603.21600.
33. Stojkoska, B.L.R.; Trivodaliev, K.V. A review of Internet of Things for smart home: Challenges and solutions. *J. Clean. Prod.* **2017**, *140*, 1454–1464.
34. Sfar, A.R.; Natalizio, E.; Challal, Y.; Chtourou, Z. A roadmap for security challenges in the Internet of Things. *Digit. Commun. Netw.* **2018**, *4*, 118–137.
35. Uhlenbeck, G.E.; Ornstein, L.S. On the Theory of the Brownian Motion. *Phys. Rev.* **1930**, *36*, 823–841.
36. Marlin P. Jones & Assoc., Inc., Lake Park, FL, USA. HC-SR501 PIR Motion Detector Datasheet. Available online: https://digsys.upc.edu/csd/chips/comp/HC_SR501.PDF (accessed on 13 January 2026).
37. InvenSense. *MPU-6000 and MPU-6050 Product Specification*; Revision 3.4; InvenSense: San Jose, CA, USA, 2013.
38. Snyder, J.P. *Map Projections—A Working Manual*; US Government Printing Office: Washington, DC, USA, 1987; Volume 1395.
39. Bettstetter, C. Smooth Mobility Models for Wireless Networks. In Proceedings of the Proceedings of IEEE VTC, Tel Aviv, Israel, 6–9 May 2001; pp. 1728–1732.

40. Arfeen, M.A.; Pawlikowski, K.; McNickle, D.; Willig, A. The role of the Weibull distribution in Internet traffic modeling. In Proceedings of the 2013 25th International Teletraffic Congress (ITC), Shanghai, China, 10–12 September 2013; pp. 1–8. <https://doi.org/10.1109/ITC.2013.6662948>.
41. Shaikh, F.K.; Zeadally, S. Energy Harvesting in Wireless Sensor Networks: A Comprehensive Review. *Renew. Sustain. Energy Rev.* **2016**, *55*, 1041–1054.
42. Leland, W.E.; Taqqu, M.S.; Willinger, W.; Wilson, D.V. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Trans. Netw.* **1994**, *2*, 1–15.
43. Lehoczky, J.P.; Ramos-Thuel, S. An optimal algorithm for scheduling soft-a-periodic tasks in fixed-priority preemptive systems. In Proceedings of the Real-Time Systems Symposium (RTSS), Phoenix, AZ, USA, 2–4 December 1992; pp. 110–123. <https://doi.org/10.1109/REAL.1992.242671>.
44. Plugge, E.; Hows, D.; Membrey, P.; Hawkins, T. *The Definitive Guide to MongoDB: A Complete Guide to Dealing with Big Data Using MongoDB*; Apress: New York, NY, USA, 2015.
45. Shelby, Z.; Koster, M.; Bormann, C.; van der Stok, P. RFC9176: Constrained RESTful Environments (CoRE) Resource Directory; IETF: Wilmington, DE, USA, 2022.
46. Elastic. Create visualizations in Kibana. Available online: <https://www.elastic.co/docs/explore-analyze/visualize> (accessed on 20 March 2025).
47. Hemminger, S. Network emulation with NetEm. In Proceedings of the Linux Conference, Canberra, Australia, 19 April 2005; pp. 18–23.
48. Intel Berkeley Research Lab. Intel Lab Data: Sensor Readings from 54 Mica2Dot Nodes. 2004. Available online: <https://db.csail.mit.edu/labdata/labdata.html> (accessed on 13 January 2026).
49. Moustafa, N. A new distributed architecture for evaluating AI-based security systems at the edge: Network TON_IoT datasets. *Sustain. Cities Soc.* **2021**, *72*, 102994.
50. Koroniotis, N.; Moustafa, N.; Sitnikova, E.; Turnbull, B. Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset. *Future Gener. Comput. Syst.* **2019**, *100*, 779–796.
51. Neto, E.C.P.; Dadkhah, S.; Ferreira, R.; Zohourian, A.; Lu, R.; Ghorbani, A.A. CICIOT2023: A real-time dataset and benchmark for large-scale attacks in IoT environment. *Sensors* **2023**, *23*, 5941.
52. Banks, A.; Briggs, E.; Borgendale, K.; Gupta, R. MQTT Version 5.0; OASIS Standard, 2019. Available online: <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html> (accessed on 28 July 2026).
53. Shelby, Z.; Hartke, K.; Bormann, C. RFC 7252; The Constrained Application Protocol (CoAP); Technical report; Universitaet Bremen: Bremen-Horn-Lehe, Germany, 2014.
54. Donenfeld, J.A. WireGuard: Next generation kernel network tunnel. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 26 February–1 March 2017.
55. McKeown, N.; Anderson, T.E.; Balakrishnan, H.; Parulkar, G.; Peterson, L.L.; Rexford, J.; Shenker, S.; Turner, J.S. High-performance switches and routers: Theory and practice. *IEEE Commun. Mag.* **1999**, *37*.
56. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley: Boston, MA, USA, 1994.
57. Martin, R.C. The Dependency Inversion Principle. *C++ Rep.* **1996**, *8*, 61–66.
58. Hartke, K. RFC 7641; Observing Resources in the Constrained Application Protocol (CoAP). Universitaet Bremen: Bremen-Horn-Lehe, Germany, 2015. <https://doi.org/10.17487/RFC7641>.
59. Quincozes, S.; Emilio, T.; Kazienko, J. MQTT protocol: Fundamentals, tools and future directions. *IEEE Lat. Am. Trans.* **2019**, *17*, 1439–1448.
60. Bormann, C.; Castellani, A.P.; Shelby, Z. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Comput.* **2012**, *16*, 62–67. <https://doi.org/10.1109/MIC.2012.29>.
61. Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M.; Ayyash, M. Internet of Things: A survey on enabling technologies, protocols, and applications. *IEEE Commun. Surv. Tutor.* **2015**, *17*, 2347–2376.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.