Benchmarking Message Brokers for IoT Edge Computing: A Comprehensive Performance Study

Tapajit Chandra Paul[1], Pawissanutt Lertpongrujikorn[1], Hai Duc Nguyen[2], and Mohsen Amini Salehi[1]
[1]High Performance Cloud Computing (HPCC) Lab, University of North Texas, USA
[2]Argonne National Laboratory and University of Chicago, USA
{tapajitchandra.paul, pawissanutt.lertpongrujikorn, mohsen.aminisalehi}@unt.edu, hai.nguyen@anl.gov

***Abstract*—Asynchronous messaging is a cornerstone of modern distributed systems, enabling decoupled communication for scalable and resilient applications. Today's message queue (MQ) ecosystem spans a wide range of designs, from high-throughput streaming platforms to lightweight protocols tailored for edge and IoT environments. Despite this diversity, choosing an appropriate MQ system remains difficult. Existing evaluations largely focus on throughput and latency on fixed hardware, while overlooking CPU and memory footprint and the effects of resource constraints, factors that are critical for edge and IoT deployments. In this paper, we present a systematic performance study of eight prominent message brokers: Mosquitto, EMQX, HiveMQ, RabbitMQ, ActiveMQ Artemis, NATS Server, Redis (Pub/Sub), and Zenoh Router. We introduce mq-bench, a unified benchmarking framework to evaluate these systems under identical conditions, scaling up to 10,000 concurrent client pairs across three VM configurations representative of edge hardware. This study reveals several interesting and sometimes counter-intuitive insights. Lightweight native brokers achieve sub-millisecond latency, while feature-rich enterprise platforms incur 2–3× higher overhead. Under high connection loads, multi-threaded brokers like NATS and Zenoh scale efficiently, whereas the widely-deployed Mosquitto saturates earlier due to its single-threaded architecture. We also find that Java-based brokers consume significantly more memory than native implementations, which has important implications for memory-constrained edge deployments. Based on these findings, we provide practical deployment guidelines that map workload requirements and resource constraints to appropriate broker choices for telemetry, streaming analytics, and IoT use cases.**

***Index Terms*—Message Brokers, Publish/Subscribe, Edge Computing, Internet of Things (IoT), Benchmarking, Performance Evaluation, Distributed Systems, MQTT.**

I. Introduction

The proliferation of Internet of Things (IoT) devices, edge computing platforms, and microservice architectures has fundamentally transformed how modern applications communicate. From smart factories monitoring thousands of sensors to connected vehicles exchanging real-time telemetry, from healthcare wearables streaming patient vitals [1] to smart city infrastructure coordinating traffic flow, the demand for reliable, low-latency messaging infrastructure continues to grow exponentially. Industry analysts project tens of billions of connected devices by 2030 [2], each generating data that must be collected, processed, and acted upon in near real-time. At the heart of these systems lies asynchronous messaging—a paradigm that enables loosely coupled components to exchange data without blocking, providing the foundation for scalable and fault-tolerant distributed applications.

Message-queue (MQ) systems serve as the backbone of this communication infrastructure, decoupling producers from consumers and enabling flexible publish/subscribe interactions across services. Unlike traditional request-response models, message queues absorb traffic bursts, enable temporal decoupling between system components, and facilitate seamless scaling of individual services. Over the past decade, the MQ landscape has diversified substantially: in addition to high-throughput streaming platforms such as Kafka, practitioners increasingly rely on lightweight brokers and protocol stacks like MQTT [3], NATS [4], Redis Pub/Sub [5], and Zenoh [6], as well as multi-protocol engines such as RabbitMQ [7] and ActiveMQ Artemis [8]. Each of these systems embodies different trade-offs in architecture, protocol semantics, and runtime implementation—from single-threaded event loops to multi-threaded actor models, from lightweight C implementations to feature-rich Java virtual machines—which can lead to sharply different performance and resource profiles in practice.

Choosing an appropriate message broker for a given deployment is inherently challenging. A broker that performs well in a well-provisioned cloud environment may not exhibit the same performance on resource-constrained edge hardware [9]. Likewise, systems optimized for high throughput can suffer from poor tail latency under bursty workloads [10], while complex broker deployments may impose memory requirements that are challenging for embedded gateways [11]. These trade-offs are especially pronounced in edge and IoT settings, where deployments span powerful fog nodes to severely constrained microcontrollers, and where network variability, power budgets, and physical space impose strict operational limits [12], [13], [14].

Although prior studies have evaluated message brokers across different environments, they typically focus on a single protocol family (e.g., MQTT [9], [15], [16]) or a limited subset of systems [17], [18], making it difficult to compare fundamentally different broker designs under comparable conditions. More importantly, existing benchmarks largely emphasize throughput and latency on fixed hardware configurations, overlooking CPU and memory footprint as well as the effects of varying resource allocations. These dimensions are critical for deploying and tuning message brokers in resource-constrained edge and IoT environments. As a result, practitioners lack consistent empirical guidance for selecting and sizing messaging infrastructure for telemetry, streaming analytics, and IoT workloads.

To address this gap, we present mq-bench, a unified bench-

marking framework that applies consistent pub/sub workloads to heterogeneous MQ systems and records throughput, latency distributions, and resource consumption in a standardized format. Using mq-bench, we conduct a head-to-head evaluation of eight widely deployed brokers—Mosquitto, EMQX, HiveMQ, RabbitMQ, ActiveMQ Artemis, NATS Server, Redis Pub/Sub, and Zenoh Router—under identical hardware and workload conditions. Our study focuses on single-node vertical scaling and resource efficiency across real-world IoT and edge deployments. We scale experiments from 500 to 10,000 concurrent client pairs and provision brokers with varying resource allocations (1–4 vCPUs, 2–8 GB RAM), spanning the strict constraints of edge devices (e.g., Raspberry Pi) to small cloud instances. By jointly measuring CPU and memory footprint alongside throughput and latency, our evaluation enables practitioners to accurately size broker deployments and assess resource headroom for target workloads.

In summary, this paper makes the following main contributions:

- We introduce **mq-bench**[1], an extensible, open-source benchmarking tool written in Rust that enables reproducible cross-protocol MQ evaluations with precise latency measurement and resource monitoring.
- We present a controlled experimental methodology on dedicated infrastructure, scaling up to 10K concurrent pub-sub pairs and reporting throughput, tail-latency percentiles (p50, p95), and CPU/memory footprint across three VM configurations representative of edge deployments.
- We provide a comparative performance characterization of eight brokers, revealing the maximum load each system can sustain under different resource constraints: NATS achieves the highest scalability (90K msg/s at 10K connections), followed by Zenoh and RabbitMQ, while single-threaded Mosquitto saturates at 45K msg/s.
- We analyze resource efficiency and show that Java-based brokers (HiveMQ, Artemis) consume 10–50× more memory than native implementations—a critical consideration for memory-constrained edge deployments.
- Based on our findings, we provide practical deployment guidelines that map workload requirements and resource constraints to appropriate broker choices, helping practitioners make informed infrastructure decisions.

The rest of the paper is organized as follows. Section II provides background on message brokers and the publish/subscribe paradigm. Section III reviews related work. Section IV describes our experimental design, including benchmark scenarios, metrics, and testbed infrastructure. Section V presents our evaluation results, and Section VI concludes with key findings and future directions.

II. Background

This section introduces the publish/subscribe messaging model and provides an overview of the messaging protocols and broker architectures evaluated in this study. We focus on single-node deployments to establish a baseline for performance comparison.

[1]The MQ-Bench source code is available at: https://github.com/hpcclab/mq-bench

A. Publish / Subscribe Architecture

Message queues enable asynchronous communication between distributed components. In the publish/subscribe model, publishers send messages to named channels called *topics*. Subscribers register interest in specific topics and receive messages published to them. A central *broker* routes messages from publishers to subscribers.

This decoupling offers several advantages. Publishers and subscribers operate independently without knowing each other's identity or location. The broker buffers messages during traffic bursts, preventing slower consumers from blocking faster producers. New publishers or subscribers can join without modifying existing components.

Figure 1 illustrates a typical pub/sub deployment in IoT systems. Devices such as an electricity meter and thermostat publish sensor data to topics (e.g., `temperature`, `intrusion_alert`), while backend systems subscribe to receive updates. The broker handles all message routing, enabling flexible scaling without direct connections between components.

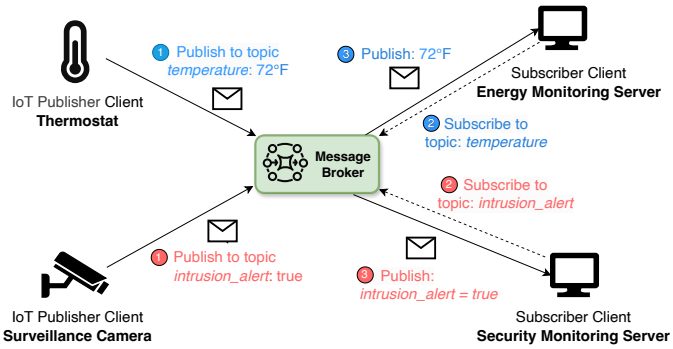


Fig. 1: Pub/sub architecture for IoT and distributed systems. Publishers send data to named topics; a central broker routes messages to subscribers, decoupling producers from consumers.

When evaluating pub/sub systems, several key performance characteristics must be considered:

- **Throughput**: Messages delivered per second under sustained load.
- **Latency**: Time from message publication to subscriber receipt.
- **Scalability**: Ability to handle increasing clients and message rates.
- **Resource efficiency**: CPU and memory consumption relative to workload.

These metrics depend on broker architecture, protocol overhead, and runtime environment. Native implementations in C, Rust, or Go typically achieve lower latency and memory usage than managed-runtime brokers in Java or Erlang, though the latter may offer richer features and easier fault tolerance.

B. Messaging Protocols

MQTT (Message Queuing Telemetry Transport) [3] is a lightweight publish/subscribe protocol for constrained devices and unreliable networks, offering three Quality of Service (QoS) levels (Table I).

AMQP (Advanced Message Queuing Protocol) [19] is an open standard for business messaging with sophisticated rout-

QoS	Guarantee	Description
0	At most once	Fire-and-forget; no acknowledgment. Fastest but may lose messages.
1	At least once	Acknowledged delivery; may produce duplicates if ACK is lost.
2	Exactly once	Four-step handshake ensures no loss or duplication. Highest overhead.

TABLE I: MQTT Quality of Service levels. Higher QoS increases reliability at the cost of latency and broker overhead.

ing (exchanges and queues) and strong reliability guarantees. We use AMQP 0-9-1 for RabbitMQ experiments.

NATS [4] is a cloud-native messaging system using a simple text-based protocol. Core NATS offers "at most once" delivery, prioritizing low latency and high throughput.

Zenoh [6] is a data-centric protocol blending publish/subscribe with distributed storage. It supports peer-to-peer communication, enabling brokerless or lightweight-router deployments that reduce network overhead.

RESP (Redis Serialization Protocol) [5] is the underlying protocol used by Redis. It provides fast pub/sub via an in-memory architecture, achieving low latency but lacking the durability of dedicated message queues.

C. Message Brokers

We evaluate a diverse set of brokers summarized in Table II, categorizing them by their primary architectural focus:

Broker	Language	Supported Protocols
Mosquitto	C	MQTT
EMQX	Erlang	MQTT
HiveMQ CE	Java	MQTT
RabbitMQ	Erlang	AMQP 0-9-1, MQTT
ActiveMQ Artemis	Java	MQTT
NATS Server	Go	NATS
Redis	C	RESP
Zenoh Router	Rust	Zenoh

TABLE II: Message brokers evaluated in this study. Only primary protocols relevant to our study are listed; some brokers support additional protocols (e.g., STOMP, OpenWire).

1) MQTT-Centric Brokers: These brokers are optimized specifically for the MQTT protocol, ranging from lightweight implementations to massive-scale engines.

Mosquitto [20] is an open-source Eclipse project written in C, using a single-threaded event-loop architecture optimized for resource-constrained hardware.

EMQX [21] is built on Erlang/OTP, leveraging the actor model for massive concurrency and fault tolerance, targeting carrier-grade MQTT deployments.

HiveMQ [22] is an enterprise-grade, Java-based MQTT broker using a non-blocking, asynchronous architecture for high scalability.

2) Enterprise Message Brokers: These systems prioritize reliability, complex routing, and protocol flexibility over raw speed, often serving as the backbone for business applications.

RabbitMQ [7] is a widely used Erlang-based broker for AMQP 0-9-1, offering flexible routing and strong delivery guarantees, often at the cost of higher latency.

ActiveMQ Artemis [8] is a high-performance, non-blocking Java message broker that supports multiple protocols, including AMQP, MQTT, and OpenWire.

3) Cloud-Native & Data-Centric Architectures: These systems depart from traditional messaging models to address specific needs like microservices communication, or edge computing.

NATS Server [4] is a CNCF (Cloud Native Computing Foundation) project written in Go, prioritizing simplicity and performance. Its "fire-and-forget" architecture minimizes overhead, making it ideal for high-frequency microservices communication.

Zenoh [6] is written in Rust with a data-centric architecture. It operates as a router or in brokerless peer-to-peer mode, representing edge-native protocols for the cloud-to-edge continuum.

Redis [5], although primarily an in-memory key-value database, is widely used for lightweight messaging. We include its Pub/Sub mechanism as a performance baseline against dedicated brokers.

III. Related Studies

MQTT-Focused Benchmarks. The majority of messaging system benchmarks concentrate on MQTT within IoT contexts [15], [16], [27], [24], [9], [23]. These studies primarily evaluate popular MQTT brokers such as Mosquitto, EMQX, and HiveMQ, examining key performance indicators including message throughput, end-to-end latency, edge deployment scenarios, QoS level behavior, and horizontal clustering capabilities. Despite their valuable contributions, these studies remain protocol-locked to MQTT and do not consider alternative messaging paradigms.

Enterprise and Cloud-Native Message Queues. A separate line of research investigates enterprise-grade message brokers [25], [26], [17], comparing systems such as RabbitMQ, ActiveMQ Artemis, and NATS using standardized benchmarking frameworks. However, these enterprise-focused studies typically omit MQTT brokers entirely, creating a gap in understanding how IoT-oriented and enterprise systems compare under equivalent conditions.

Emerging Protocols and Data-Centric Middleware. Recent work has begun exploring newer messaging paradigms beyond traditional broker architectures. Liang et al. [18] studied Zenoh in Industrial IoT settings, highlighting potential latency benefits relative to MQTT-based systems. In addition, Redis Pub/Sub remains relatively understudied in academic literature despite its widespread adoption in production systems for lightweight messaging.

Gap and Contribution. As Table III summarizes, no existing study comprehensively evaluates MQTT brokers (Mosquitto, EMQX, HiveMQ), multi-protocol message queues (RabbitMQ, ActiveMQ Artemis), cloud-native systems (NATS), in-memory pub/sub transports (Redis), and data-centric middleware (Zenoh) within a unified experimental framework. Our work addresses this gap through mq-bench, a Rust-based benchmarking tool that enables reproducible, cross-protocol performance comparisons using consistent workload patterns, measurement methodologies, and resource utilization metrics across eight broker systems.

References	Protocol	Message Queues								Metrics					
		Mosq.	EMQX	HiveMQ	Artemis	RabbitMQ	NATS	Redis	Zenoh	Latency	Throughput	Resource	Scalability	QoS	Other
Liang et al. (2023) [18]	M, Z	✓							✓	✓	✓				
Koziolek et al. (2020) [9]	M		✓	✓						✓	✓	✓			✓
Dizdarevic et al. (2023) [16]	M	✓	✓	✓	✓					✓					✓
Mishra (2018) [23]	M	✓		✓	✓					✓	✓			✓	✓
Mishra et al. (2021) [15]	M	✓	✓	✓						✓	✓	✓		✓	
Pazos et al. (2024) [24]	M	✓	✓	✓						✓				✓	✓
Ibrahim et al. (2025) [25]	A, N				✓	✓	✓			✓	✓		✓		
Maharjan et al. (2023) [26]	A, R				✓	✓		✓		✓	✓				
Chy et al. (2023) [17]	A				✓					✓	✓	✓			✓
Our Study	M, A, N, R, Z	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

TABLE III: Comparison of message queue systems and evaluation metrics in existing studies. Protocol: M=MQTT, A=AMQP, N=NATS, R=Redis, Z=Zenoh.

IV. Experimental Design

Message brokers are often deployed in resource-constrained edge environments where they must handle high client loads and varying message sizes. Our evaluation focuses on understanding how different broker architectures perform under these challenging conditions. Specifically, we aim to answer:

- **Latency under varying payloads**: How does message size affect end-to-end delivery latency? IoT applications range from small sensor readings to bulk data transfers, and brokers must handle this diversity efficiently.
- **Throughput under high client load**: What is the maximum number of concurrent clients each broker can sustain? As deployments scale, brokers face increasing connection counts and message rates.
- **Resource efficiency under constraints**: How do brokers perform when CPU and memory are limited? Edge devices often have strict resource budgets, making efficiency critical.
- **Reliability under network instability**: How do MQTT brokers maintain QoS guarantees when network conditions are unstable?

To address these objectives, we design three experiments which are detailed in the (Section IV-A). We describe the metrics collected in Section IV-B, broker deployment configurations in Section IV-C, workload generation strategy in Section IV-D, the benchmark tool in Section IV-F, and testbed infrastructure in Section IV-E.

A. Benchmark Scenarios

We design three experiments to evaluate broker performance. Figure 2 illustrates the test scenarios.

1) Experiment 1: Latency vs. Payload Size: This experiment characterizes how message size affects end-to-end latency. We measure median (p50) and tail (p95) latency at three payload sizes: 1 KB (sensor telemetry), 16 KB (IoT aggregation), and 1 MB (bulk data transfer). Table IV summarizes the parameters.

Parameter	Value
Number of pub-sub pairs	10
Message rate (per publisher)	10 msg/s
Payload sizes	1 KB, 16 KB, 1 MB
Test duration	120 seconds
Warmup period	60 seconds
QoS level	0 (best-effort)

TABLE IV: Latency vs. Payload Size parameters.

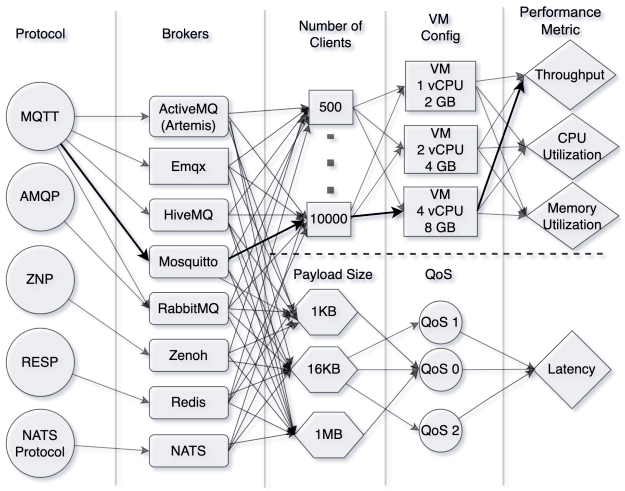


Fig. 2: Experimental scenarios evaluated in this study. Each path from start to end represents one test scenario. For instance, the solid black path represents the scenario in which MQTT–Mosquitto is evaluated with 10,000 clients on a VM with 4-vCPU, 8-GB RAM configuration while measuring throughput.

2) Experiment 2: Throughput and Resource Utilization: This experiment measures broker capacity and resource consumption under increasing client load. We progressively increase the number of pub-sub pairs until saturation. Table V summarizes the parameters.

We measure throughput, CPU utilization, and memory consumption at each scale point. Saturation is indicated by throughput dropping below the offered load. We repeat measurements across three VM configurations (Section IV-C2) to characterize vertical scaling efficiency.

In addition to the 1-to-1 topology, we repeat this experiment with fanout topology.

Parameter	Value
Number of pub-sub pairs	500, 1000, 2000, ..., 10,000
Message rate (per publisher)	10 msg/s
Offered load	#pairs × rate
Payload size	1 KB
Test duration	120 seconds
Warmup period	60 seconds
QoS level	0 (best-effort)

TABLE V: Throughput and Resource Utilization parameters.

3) Experiment 3: QoS Reliability Under Network Failures: This experiment evaluates how MQTT brokers maintain Quality of Service guarantees when subscribers experience network

failures. Unlike controlled disconnections, network failures via TCP RST packets simulate realistic cloud and edge scenarios where connections drop unexpectedly. Table VI summarizes the parameters.

With MTTF=30s and MTTR=5s over a 180-second duration, each test experiences approximately 5–6 network failures with a total downtime of 25–30 seconds (approximately 85% uptime). We test only the five MQTT-native brokers (Mosquitto, EMQX, HiveMQ, RabbitMQ, and Artemis) as NATS, Redis, and Zenoh use different QoS semantics. For each QoS level, we measure message loss percentage and end-to-end latency to characterize both reliability and the latency impact of different QoS levels.

Parameter	Value
Number of pub-sub pairs	10
Message rate (per publisher)	10 msg/s
Payload size	1 KB
Test duration	180 seconds
Mean time to failure (MTTF)	30 seconds
Mean time to recovery (MTTR)	5 seconds

TABLE VI: QoS Reliability Under Network Failures parameters.

B. Measurement Metrics

We collect the following metrics during each experiment:

- **Latency**: End-to-end message latency measured from send timestamp (embedded in message header) to receive timestamp (recorded by subscriber). We report the minimum, median (p50) and tail-latency percentile (p95) to capture both typical and worst-case behavior.
- **Throughput**: The number of messages successfully delivered per second, calculated as Throughput = received_count/duration (message per second).
- **CPU Utilization**: Broker container CPU usage as a percentage of allocated vCPUs, sampled via Docker [28] Stats API at 1-second intervals.
- **Memory Footprint**: Broker container resident set size (RSS) in megabytes, sampled via Docker Stats API.

C. Broker Deployment Configurations

1) Broker Selection: We evaluate the eight message broker implementations summarized in Table II (Section II), covering lightweight MQTT brokers (Mosquitto), enterprise-grade MQTT platforms (EMQX, HiveMQ), multi-protocol message queues (RabbitMQ, ActiveMQ Artemis), cloud-native systems (NATS), in-memory pub/sub (Redis), and data-centric middleware (Zenoh). For Zenoh, we strictly configure it in client/router mode rather than peer-to-peer mode to ensure a fair comparison with other broker-based systems. All brokers are deployed as single-node instances without clustering to isolate core broker performance from distributed coordination overhead. To ensure fair comparison, we configure each broker to fully utilize the available CPU resources by tuning thread pool sizes and worker counts if needed to match the number of allocated vCPUs. The exception is Mosquitto, which employs a single-threaded event-loop architecture and cannot be parallelized across multiple cores.

2) Container and VM Configuration: Each broker runs inside a Docker container [29], [30] hosted within a KVM-based virtual machine on the Broker Execution Machine. Docker containers are configured with high file descriptor limits (`nofile: 300000`) to support large numbers of concurrent connections. For scalability experiments, we use three VM configurations that mirror typical edge computing hardware such as the Raspberry Pi [**?**]: (1) **1 vCPU / 2 GB RAM**: Constrained edge devices such as IoT gateways, (2) **2 vCPU / 4 GB RAM**: Mid-range edge hardware such as small industrial PCs. (3) **4 vCPU / 8 GB RAM**: Higher-capacity fog nodes and small servers.

3) Protocol Mappings: Each broker is tested using its native protocol: MQTT for Mosquitto, EMQX, and HiveMQ; NATS for NATS Server; RESP for Redis; and Zenoh for Zenoh Router. ActiveMQ Artemis is evaluated via its MQTT interface for protocol-level consistency. RabbitMQ is evaluated using both MQTT and AMQP 0-9-1 in separate runs.

D. Workload Generation Strategy

1) Communication Topology: We adopt a 1-to-1 (pair) topology where each publisher is paired with exactly one subscriber on a dedicated topic. This design provides isolation (no inter-client interference), enables scalability testing (stress connection handling and routing), ensures reproducibility (symmetric setup), and establishes a baseline for future fan-in/fan-out studies. For the throughput experiment, we additionally evaluate a fanout topology on the 4 vCPU, 8 GB RAM VM, where a single publisher broadcasts to N subscribers on a shared topic.

2) QoS Configuration: First two experiments use QoS level 0 (at-most-once delivery) for MQTT-based brokers and equivalent best-effort semantics for other protocols. This ensures fair comparison across protocols and measures peak broker capacity. The third experiment, however, evaluates all three MQTT QoS levels.

E. Testbed Overview

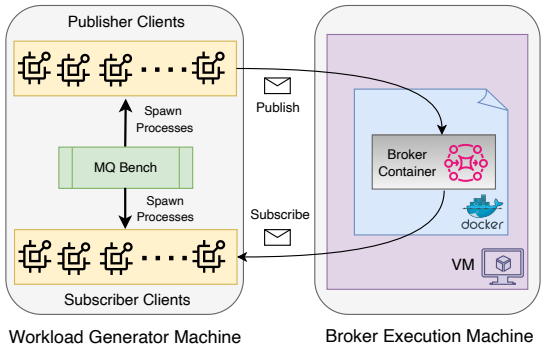


Fig. 3: Experimental testbed architecture. The Workload Generator Machine runs benchmarking tool (mq-bench) to spawn publisher/-subscriber clients that send and receive messages to/from the broker container hosted on the Broker Execution Machine within a VM.

Figure 3 illustrates the architecture of our experimental testbed, which consists of two dedicated bare-metal nodes provisioned from Chameleon Cloud [31] (node type: `compute_icelake_r650`, image: `CC-Ubuntu22.04`), connected via a high-speed private network.

Workload Generator Machine. This machine hosts the mq-bench benchmarking tool and orchestrates the entire experiment lifecycle. At the start of each iteration, the tool brings up the broker container on the remote Broker Execution Machine. It then spawns concurrent Publisher and Subscriber clients that generate load against the broker according to the configured workload parameters. During execution, mq-bench collects all transmitted and received messages along with their timestamps, which are used to calculate performance metrics including throughput, latency, and message loss. Upon completion, the tool brings the broker container down before proceeding to the next iteration. A detailed description of mq-bench's architecture is provided in Section IV-F.

Broker Execution Machine. This machine provides an isolated execution environment for the message broker under test. The broker runs within a Docker container [28], [30] inside a KVM-based virtual machine, ensuring consistent resource allocation and minimal interference from host processes. The hypervisor layer hosts three pre-configured VM instances as mentioned in the Section IV-C2 to evaluate broker scalability under varying computational resources. A resource monitoring module continuously tracks CPU and memory footprint of the broker container in real-time throughout each experiment iteration.

F. Benchmark Tool

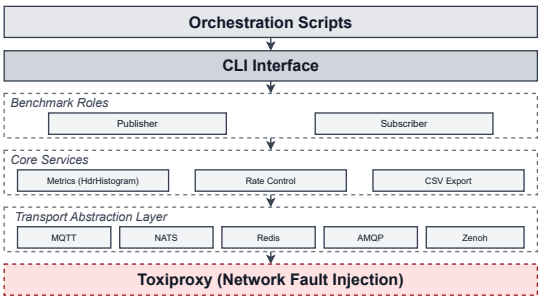


Fig. 4: Architecture of mq-bench. Orchestration scripts invoke the CLI to spawn Publisher and Subscriber roles. Core services handle metrics collection, rate limiting, and result export.

As introduced in Section IV-E, we developed **mq-bench**, a high-performance benchmarking tool that resides on the workload generator machine (Figure 3) and orchestrates the entire experiment lifecycle. Developed in Rust using the Tokio asynchronous runtime, mq-bench utilizes a transport abstraction layer to offer a unified load-generation interface across various messaging protocols. Figure 4 depicts the tool's architecture.

1) Architecture: The benchmark tool consists of pluggable transport adapters for MQTT (via `rumqttc`), NATS (via `async-nats`), Redis (via `redis-rs`), AMQP (via `lapin`), and Zenoh (native SDK). Each adapter implements a common `Transport` trait supporting subscribe, publish, and connection management operations.

2) Latency Measurement: Each message includes a 24-byte binary header containing a 64-bit send timestamp in UNIX nanoseconds. Upon receiving a message, subscribers extract this timestamp and compute end-to-end latency as latency $= t_{\text{recv}} - t_{\text{send}}$, where t_{recv} is the local receive time. Clock synchronization between publisher and subscriber processes is ensured by running both on the same workload generator machine. We record latency values in nanoseconds and aggregate samples to report percentiles, average, and standard deviation.

3) Throughput Measurement: Publishers operate in open-loop mode using a token-bucket rate controller, emitting messages at a fixed rate (e.g., 10 msg/s per publisher) independent of broker acknowledgments or backpressure. This design ensures a consistently offered load regardless of broker response times, accurately revealing saturation behavior when brokers cannot keep pace. To accurately measure throughput, we identify a stable period during each experiment run. A post-processing script analyzes the connection logs and detects when all expected clients have successfully connected to the broker. In case of broker overload and the broker cannot accept target client connections, the script instead identifies when the connection count reaches saturation. Throughput is then calculated only from messages received during this stable period, excluding the initial connection phase. This approach ensures that measurements reflect steady-state broker performance rather than transient startup behavior.

4) Network Fault Injection: For resilience testing, mq-bench integrates with Toxiproxy [32], a TCP proxy that intercepts connections and injects configurable faults such as latency, bandwidth limits, or connection resets. Failure timing follows an exponential distribution, modeling Poisson failure arrivals common in reliability engineering. The MTTF parameter controls the average interval between failures, while MTTR defines the reconnection delay. Upon proxy recovery, mq-bench automatically reconnects subscribers using the same client ID, enabling MQTT session resumption and message redelivery for QoS 1/2.

V. Experimental Results

This section presents the experimental results from our benchmark study. We evaluate the brokers across three dimensions: latency under varying payload sizes, throughput under client scaling, and reliability under network disruptions. For latency experiments, we use a 4 vCPU, 8GB RAM VM to ensure sufficient headroom and isolate protocol-level differences. For throughput experiments, we vary VM configurations (1, 2, and 4 vCPUs) to observe how brokers scale with available resources.

A. Latency vs Payload Size

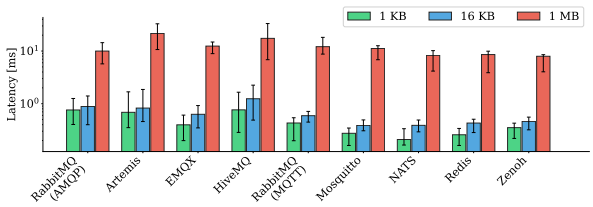


Fig. 5: Median latency with error bars (min to P95) across payload sizes. The y-axis uses a logarithmic scale.

Figure 5 presents end-to-end latency measurements across three payload sizes: 1KB, 16KB, and 1MB. The bar height

represents median (P50) latency, with error bars extending from the minimum observed latency to P95.

For small 1KB payloads, native brokers achieve sub-millisecond latencies. NATS leads with the lowest median at 0.21ms, followed closely by Redis (0.26ms) and Mosquitto (0.27ms). Zenoh also performs well at 0.35ms. These brokers benefit from lightweight protocols and efficient memory handling in their native implementations. The managed-runtime brokers show higher latencies: EMQX and RabbitMQ-MQTT both achieve around 0.40–0.43ms, while Artemis and HiveMQ exhibit notably higher medians of 0.69ms and 0.76ms respectively. The JVM-based brokers (HiveMQ, Artemis) show wider variance, with HiveMQ's P95 reaching 1.65ms—roughly 5× higher than NATS.

At 16KB, all brokers show modest latency increases as expected from the larger serialization and transmission overhead. NATS maintains the lowest median at 0.39ms, with Mosquitto close behind at 0.39ms. The relative ordering remains consistent, though the gap between native and managed-runtime brokers widens slightly. HiveMQ's median increases to 1.24ms with P95 at 2.25ms, reflecting the JVM's additional overhead when processing larger payloads.

For 1MB payloads, serialization and network transfer costs dominate the latency profile. NATS achieves the lowest median at 8.27ms, followed by Zenoh (7.96ms) and Redis (8.58ms). Interestingly, AMQP-RabbitMQ outperforms MQTT-RabbitMQ at this payload size (9.99ms vs 12.11ms), suggesting AMQP's binary framing handles large messages more efficiently. Mosquitto shows higher latency (11.18ms) than expected for a native broker, likely due to its single-threaded architecture becoming a bottleneck during large payload processing. The JVM-based brokers struggle significantly: HiveMQ and Artemis exhibit medians of 17.41ms and 21.60ms respectively, with P95 latencies exceeding 32ms. This 2–3× latency penalty reflects garbage collection overhead and object serialization costs in managed runtimes when handling large messages. EMQX performs better among managed brokers at 12.42ms, benefiting from Erlang's efficient binary handling.

Takeaway: *For small messages, all brokers deliver similar latencies. As payload size grows, native brokers stay fast while managed-runtime brokers slow down significantly.*

B. Throughput vs Client Scaling

This section presents throughput, CPU, and memory footprint results across three VM configurations. With each publisher sending 10 msg/s, target throughput is pairs × 10 (e.g., 10K msg/s for 1000 pairs). Figure 6 presents all results in a consolidated view. Note that RabbitMQ-AMQP is excluded from the throughput graphs as it failed early in each configuration—after 500 pairs for all the VM configurations. This early failure is attributed to AMQP's heavier protocol overhead compared to MQTT, which requires more resources for connection management and channel multiplexing under high concurrency. Consequently, we were unable to collect throughput data for RabbitMQ-AMQP at higher client pair counts.

1) 1 vCPU, 2GB RAM: On our most constrained setup, up to 1000 pairs all brokers keep up without issue. Performance differences emerge as we push higher. Zenoh and NATS pull ahead of the pack—Zenoh sustains an impressive 50.8K msg/s at 5000 pairs, while NATS tops out around 34K msg/s. Both degrade gracefully rather than crashing when pushed beyond their limits. Redis holds its own up to about 25K msg/s before hitting CPU saturation, keeping memory usage minimal at just 62–75MB even under heavy load. Mosquitto's single-threaded architecture caps it at around 10K msg/s, but it remains stable within that range.

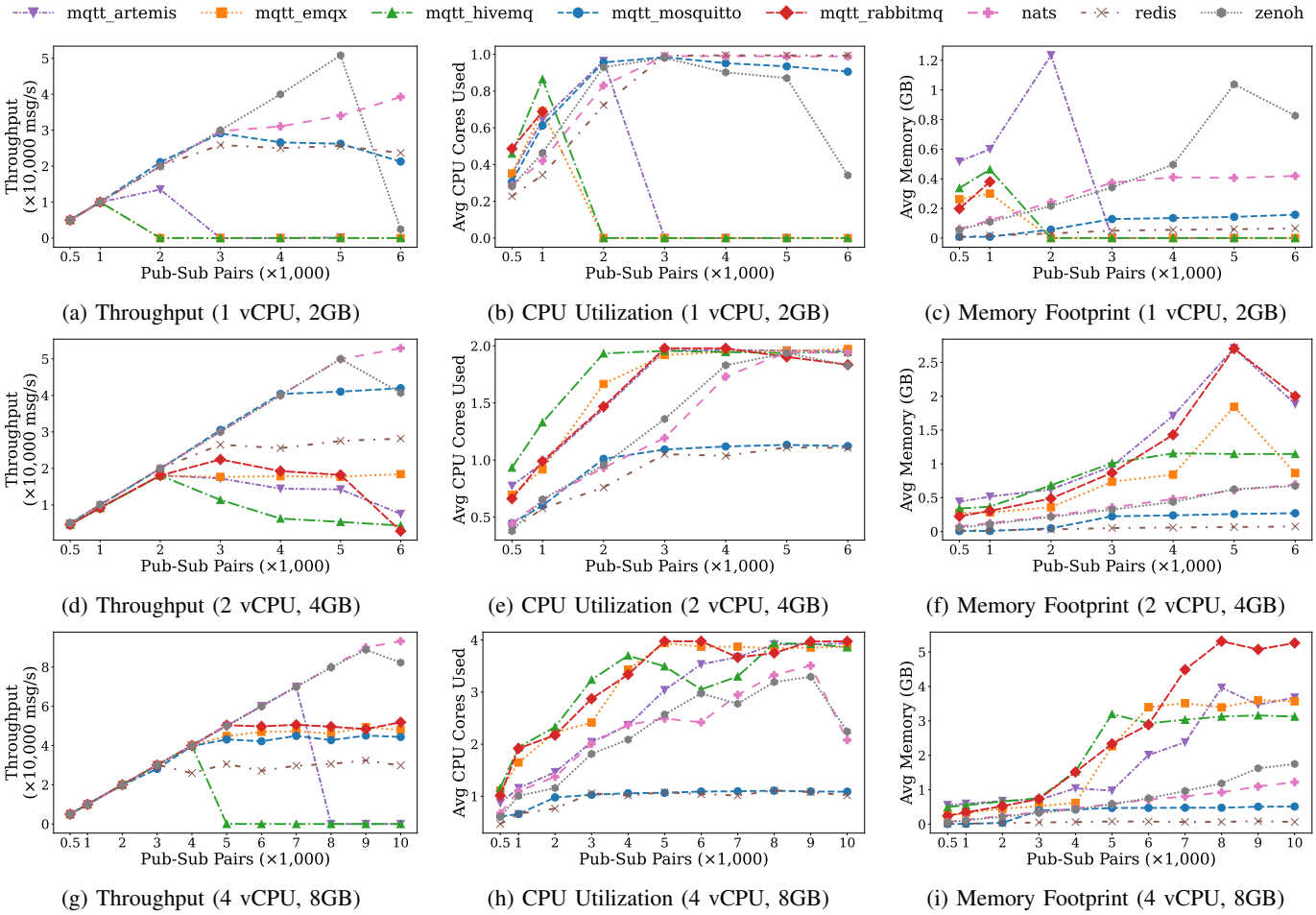
The JVM-based brokers—HiveMQ and Artemis—struggle on this limited hardware. HiveMQ stalls beyond 10K msg/s, and Artemis fails entirely past 30K msg/s. Both consume significant memory (487MB and 1.3GB respectively), reflecting the overhead of garbage collection under pressure, and are susceptible to OOM kills in this constrained environment. EMQX hits a wall at 20K msg/s, suggesting the Erlang VM needs more headroom to operate reliably. In some cases, brokers crash shortly after clients begin connecting. When this occurs, all metrics (throughput, CPU, and memory) drop to near-zero in the graphs since we report averages over the entire experiment duration. For instance, HiveMQ & EMQX crashes at 2000 pairs and Artemis at 3000 pairs. This indicates that brokers written in systems languages like C, Rust, and Go deliver better throughput per resource on constrained hardware than those running on managed runtimes.

2) 2 vCPU, 4GB RAM: The additional resources provide substantial improvements for multi-threaded brokers, while single-threaded implementations show more modest gains. Zenoh and NATS benefit significantly from the second core. Zenoh sustains approximately 50K msg/s at 5000 pairs and gracefully degrades to 41K msg/s at 6000 pairs, compared to crashing in VM1. NATS scales to 53K msg/s at 6000 pairs—a 56% improvement over VM1's 34K msg/s. Both effectively utilize both cores, with moderate memory footprints of 656MB and 728MB respectively.

Redis and Mosquitto, constrained by their single-threaded architectures, show incremental gains. Redis improves from 26K to 28K msg/s and Mosquitto from 26K to 40K msg/s as additional memory helps reduce backpressure. Redis remains memory-efficient at 77–87MB, while Mosquitto uses 284–308MB.

The managed-runtime brokers benefit more noticeably from the doubled memory. EMQX, which failed beyond 1000 pairs in VM1, now sustains 18K msg/s up to 2000 pairs before stalling, consuming 900MB–1.3GB. HiveMQ recovers from complete failure in VM1 to achieve 18K msg/s at 2000 pairs. However, it degrades severely to 4.3K msg/s at 6000 pairs with memory usage of 1.2–1.7GB. RabbitMQ-MQTT reaches 22K msg/s at 3000 pairs before declining to 2.8K msg/s at 6000 pairs, with memory reaching 2.1GB. Artemis degrades from 18K to 7.5K msg/s as load increases, with memory approaching 2GB. Overall, the doubled resources reduce OOM pressure for garbage-collected runtimes, while native multi-threaded implementations gain from parallel execution.

3) 4 vCPU, 8GB RAM: With four cores available, multi-threaded brokers scale significantly compared to the VM2. We can scale till 10000 client pairs for most brokers, although



(a) Throughput (1 vCPU, 2GB) (b) CPU Utilization (1 vCPU, 2GB) (c) Memory Footprint (1 vCPU, 2GB)

(d) Throughput (2 vCPU, 4GB) (e) CPU Utilization (2 vCPU, 4GB) (f) Memory Footprint (2 vCPU, 4GB)

(g) Throughput (4 vCPU, 8GB) (h) CPU Utilization (4 vCPU, 8GB) (i) Memory Footprint (4 vCPU, 8GB)

Fig. 6: Throughput and resource utilization as client pairs scale across three VM configurations. Each row represents a VM configuration (top: 1 vCPU/2GB, middle: 2 vCPU/4GB, bottom: 4 vCPU/8GB). Columns show throughput (msg/s), CPU utilization (cores used), and memory footprint (MB) respectively.

the performance varied significantly. Zenoh can sustain till the 9000 pairs having approximately the perfect throughput of 90K msg/s at 9000 pairs before degrading to 82K msg/s at 10000 pairs. NATS also reaches 90K msg/s at 9000 pairs. Both utilize 3.3–3.5 cores on average with memory footprints of 1.1–1.7GB.

Redis and Mosquitto, limited by their single-threaded architectures, show minimal gains compared to 2 vCPUs. Redis remains around 30K msg/s, while Mosquitto improves modestly to 45K msg/s. Redis maintains its memory efficiency at 66–90MB, while Mosquitto uses 500–620MB.

The managed-runtime brokers benefit most dramatically from the quadrupled resources. EMQX improves from 18K msg/s in VM2 to 47K msg/s—a 161% gain—utilizing 3.4–3.9 cores and 650MB–3.7GB memory. Artemis shows the largest improvement, jumping from 18K to 70K msg/s at 7000 pairs before failing at 8000 pairs. RabbitMQ-MQTT sustains 50K msg/s at 5000–6000 pairs, more than doubling its VM2 performance of 22K msg/s. HiveMQ recovers from severe degradation in VM2 to achieve 40K msg/s at 4000 pairs, though it still fails beyond 5000 pairs. These brokers consume substantial memory under load, ranging from 1.6GB to 5.5GB. The results confirm that JVM and Erlang-based brokers require significant resources to perform competitively, but can match or exceed native implementations when adequately provisioned.

Takeaway: *How well a broker scales depends on its threading model and runtime. Multi-threaded native brokers benefit most from extra cores, single-threaded ones hit a ceiling, and managed-runtime brokers need generous resources to keep up.*

4) Fanout Topology: We additionally evaluate a fanout topology on the 4 vCPU, 8GB RAM VM: a single publisher sends 100 msg/s to a shared topic while N subscribers each receive every message, yielding an aggregate target of $N \times 100$ msg/s. This isolates the broker's dispatch path from its ingestion cost. Note that we use 100 msg/s here instead of the 10 msg/s rate used in the 1-to-1 topology. Because there is only one publisher in the fanout setup, keeping 10 msg/s would place negligible load on the brokers–nearly all of them performed perfectly under that rate–so we increased the message rate to 100 msg/s to stress the dispatch path and expose saturation behavior. Results are shown in Figure 7.

Because of this rate difference, raw fanout numbers should

not be directly compared with 1-to-1 results. The most striking result is Zenoh's dominance: it sustains up to 850K msg/s at 10,000 subscribers while CPU plateaus at ∼2 cores–well below the 4-core ceiling–suggesting an efficient broadcast dispatch path that avoids per-subscriber overhead. This is a meaningful reversal from 1-to-1, where Zenoh and NATS performed comparably. In fanout, NATS scales well to 4,000 subscribers (∼400K msg/s) but then collapses to 167K msg/s at 10,000 despite saturating all four cores, indicating that its dispatch cost scales linearly with subscriber count. Most other brokers–EMQX, Redis, Mosquitto, and HiveMQ–saturate early at ∼100K msg/s regardless of subscriber count, roughly matching their 1-to-1 ceiling. RabbitMQ shows limited scaling in both topologies, while Artemis fails at higher client counts under fanout. CPU and memory usage figures are available in the mq-bench GitHub repository.

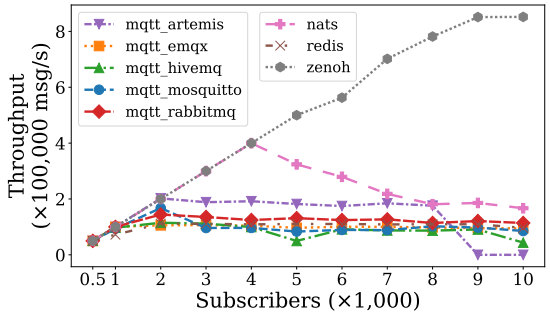


Fig. 7: Aggregate subscriber throughput under the fanout topology (1 publisher, N subscribers) on the 4 vCPU, 8GB RAM configuration.

Takeaway: *Performance trends from point-to-point topologies generally carry over to fanout, but topology can shift bottlenecks and flip relative rankings: Zenoh, which matched NATS in 1-to-1, dominates in fanout with near-linear throughput scaling and modest CPU usage, while NATS collapses beyond 4,000 subscribers despite saturating all four cores.*

C. Reliability and Latency Under Network Disruptions

While the previous experiments measured performance under stable conditions, real-world deployments must handle network instability. We evaluate QoS guarantees under TCP RST fault injection, simulating abrupt subscriber disconnections.

Message Reliability Under Network Failures. Each test transmitted 18,200 messages over 180 seconds while experiencing 5–6 network failure events. At QoS 0 (at-most-once), all brokers exhibit approximately 6.3–6.6% message loss, corresponding to 1,150–1,200 lost messages. The minor variation across brokers stems from the randomness in failure timing rather than implementation differences, as QoS 0 provides no buffering or redelivery mechanism.

At QoS 1 (at-least-once) and QoS 2 (exactly-once), all five brokers achieve 0% message loss. This demonstrates that persistent session support (`clean_session=false`) combined with stable client identifiers enables reliable message delivery despite multiple network failures. During disconnection periods, brokers buffer messages for offline subscribers and deliver them upon reconnection.

End-to-End Latency Under Failures. Figure 8 presents P50 latency with error bars across QoS levels. The key observation is the dramatic increase in latency variance for QoS 1 and 2 compared to QoS 0.

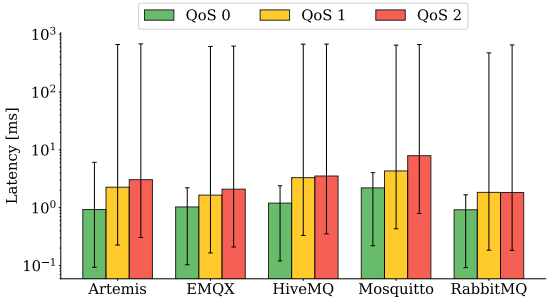


Fig. 8: Median latency with error bars (min to P95) by broker and QoS level under network failures.

For QoS 0, all brokers maintain low and consistent latencies with P50 under 2.5ms and P99 under 30ms. Messages are delivered immediately without acknowledgment overhead, resulting in minimal variance.

For QoS 1 and 2, broker architectural differences become apparent. EMQX and RabbitMQ achieve the lowest P50 latencies (1.65–2.1ms), comparable to their QoS 0 performance. Both benefit from lightweight concurrency models: EMQX's Erlang processes and RabbitMQ's actor-based message passing handle acknowledgments asynchronously without blocking the main message path. Artemis and HiveMQ show moderate latencies (2.3–3.5ms), as their JVM thread pools manage acknowledgment state efficiently despite higher per-thread overhead. Mosquitto exhibits the highest latencies, increasing from 2.20ms at QoS 0 to 7.93ms at QoS 2—a 3.6× penalty. This stems from its single-threaded architecture, which must serialize message delivery with acknowledgment processing, blocking new messages until QoS handshakes complete.

Tail latencies tell a different story: P95 and P99 spike to 475–665ms and ∼3,000ms respectively across all brokers uniformly. This reflects message buffering during the 5-second MTTR window, with reconnection overhead dominating rather than broker-specific factors.

Takeaway: *Higher QoS levels add acknowledgment overhead that multi-threaded brokers handle in the background with little latency cost, while single-threaded brokers must process acknowledgments one at a time, slowing delivery down.*

VI. Conclusion and Future Works

This paper presented mq-bench, a unified benchmarking framework for evaluating message queue systems across heterogeneous protocols and broker architectures. We evaluated eight brokers–Mosquitto, EMQX, HiveMQ, RabbitMQ, ActiveMQ Artemis, NATS Server, Redis, and Zenoh–under identical hardware and workload conditions.

Our evaluation reveals several key findings. Native brokers written in systems languages (C, Rust, Go) achieve lower latency than managed-runtime brokers, with JVM-based brokers showing 2–3× higher latency for large payloads. Multi-threaded native brokers (Zenoh, NATS) scale

well with additional CPU cores, reaching up to 90K msg/s, while single-threaded brokers (Redis, Mosquitto) remain stable but do not benefit from parallelism. Memory efficiency varies significantly–Redis maintains minimal usage (66–90MB) while JVM-based brokers consume 10–50× more, an important consideration for edge deployments.

Our study also uncovered some counterintuitive findings. Managed-runtime brokers can match or exceed native performance when given adequate resources–Artemis achieved 70K msg/s on 4 vCPU, higher than Redis and Mosquitto. Conversely, Mosquitto showed higher latency than expected for large payloads due to its single-threaded architecture.

Future Work. We plan to extend mq-bench to evaluate distributed multi-node deployments, additional workload patterns such as request-reply semantics, and diverse network conditions typical of wide-area IoT deployments.

Based on our findings, we offer the following guidelines for practitioners:

- **Constrained edge devices:** Single-threaded native brokers (Mosquitto, Redis) offer stability and minimal memory footprint, making them a good fit for resource-constrained deployments.
- **Mid-range edge hardware:** Multi-threaded native brokers (Zenoh, NATS) provide better throughput scaling, well-suited for moderately provisioned edge nodes.
- **Latency-sensitive applications:** Native brokers achieve lower latency, especially for larger payloads, and are preferable for time-critical workloads.
- **Feature-rich deployments:** Managed-runtime brokers (RabbitMQ, Artemis) offer richer features and can deliver competitive performance with sufficient resources.
- **Reliable delivery:** MQTT brokers with QoS 1/2 support reliable delivery; multi-threaded implementations offer lower latency overhead.

Acknowledgment

We would like to thank anonymous reviewers for their constructive feedback; and Chameleon Cloud for providing resources. This project is supported by National Science Foundation (NSF) through CNS CAREER Award# 2419588.

References

[1] M. Wu, "Wearable technology applications in healthcare: a literature review," *On-Line Journal of Nursing Informatics*, vol. 23, no. 3, 2019.

[2] IoT Analytics, "State of iot 2024: Number of connected iot devices growing 13% to 18.8 billion globally," https://iot-analytics.com/number-connected-iot-devices/, 2024, online; Accessed on 1 Dec. 2025.

[3] OASIS, "MQTT Version 5.0," https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, online; Accessed on 1 Dec. 2025.

[4] Synadia, "NATS Documentation," https://docs.nats.io/, online; Accessed on 1 Dec. 2025.

[5] Redis, "Redis Pub/Sub," https://redis.io/docs/interact/pubsub/, online; Accessed on 1 Dec. 2025.

[6] Z. Technology, "Zenoh: Zero Overhead Network Protocol," https://zenoh.io/, online; Accessed on 1 Dec. 2025.

[7] VMware, "RabbitMQ: One broker to queue them all," https://www.rabbitmq.com/, online; Accessed on 1 Dec. 2025.

[8] A. S. Foundation, "ActiveMQ Artemis," https://activemq.apache.org/components/artemis/, online; Accessed on 1 Dec. 2025.

[9] H. Koziolek, S. Grüner, and J. Rückert, "A comparison of mqtt brokers for distributed iot edge computing," in *European Conference on Software Architecture*. Springer, 2020, pp. 352–368.

[10] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[11] J. E. Luzuriaga, J. C. Cano, C. Calafate, P. Manzoni, M. Perez, and P. Boronat, "Handling mobility in iot applications using the mqtt protocol," in *2015 Internet Technologies and Applications (ITA)*. IEEE, 2015, pp. 245–250.

[12] N. Fahad, M. Touhiduzzaman, and E. Bulut, "Ensemble learning based wifi sensing using spatially distributed tx-rx links," in *2025 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2025, pp. 606–611.

[13] P. Lertpongrujikorn, J. Kwon, H. D. Nguyen, and M. Amini_Salehi, "Edgeweaver: Accelerating iot application development across edge-cloud continuum," 2025.

[14] P. Lertpongrujikorn, M. Amini Salehi, and T. C. Paul, "Object-as-a-service (oaas): Streamlining cloud-native application development for edge-cloud continuum," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) Tutorials*, May 2026, accepted.

[15] B. Mishra, B. Mishra, and A. Kertesz, "Stress-testing mqtt brokers: A comparative analysis of performance measurements. energies, 14, article 5817," 2021.

[16] J. Dizdarević, M. Michalke, A. Jukan, X. Masip-Bruin, and F. D'Andria, "Benchmarking performance of various mqtt broker implementations in a compute continuum," in *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2024, pp. 357–366.

[17] M. S. H. Chy, M. A. R. Arju, S. M. Tella, and T. Cerny, "Comparative evaluation of java virtual machine-based message queue services: A study on kafka, artemis, pulsar, and rocketmq," *Electronics*, vol. 12, no. 23, p. 4792, 2023.

[18] W.-Y. Liang, Y. Yuan, and H.-J. Lin, "A performance study on the throughput and latency of zenoh, mqtt, kafka, and dds," *arXiv preprint arXiv:2303.09419*, 2023.

[19] OASIS, "Advanced Message Queuing Protocol (AMQP) Version 1.0," http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf, online; Accessed on 1 Dec. 2025.

[20] E. Foundation, "Eclipse Mosquitto," https://mosquitto.org/, online; Accessed on 1 Dec. 2025.

[21] E. Technologies, "EMQX: The Unified MQTT Platform for AI & IoT Data Streaming," https://www.emqx.io/, online; Accessed on 1 Dec. 2025.

[22] HiveMQ, "HiveMQ: Enterprise MQTT Broker," https://www.hivemq.com/, online; Accessed on 1 Dec. 2025.

[23] B. Mishra, "Performance evaluation of mqtt broker servers," in *International Conference on Computational Science and Its Applications*. Springer, 2018, pp. 599–609.

[24] F. Pazos, "Performance evaluation of mqtt broker servers deployed in the cloud," *Electronic Journal of SADIO*, vol. 23, 2024.

[25] A. G. Ibrahim, R. P. Lopes, J. Rufino, and P. Leitão, "On the impact of message brokers implementations in the choreography of microservices," in *International Conference on Optimization, Learning Algorithms and Applications*. Springer, 2025, pp. 3–17.

[26] R. Maharjan, M. S. H. Chy, M. A. Arju, and T. Cerny, "Benchmarking message queues," in *Telecom*, vol. 4, no. 2. MDPI, 2023, pp. 298–312.

[27] M. Kashyap, A. K. Dev, and V. Sharma, "Implementation and analysis of emqx broker for mqtt protocol in the internet of things," *e-Prime-Advances in Electrical Engineering, Electronics and Energy*, vol. 10, p. 100846, 2024.

[28] Docker Inc., "Docker: Accelerated container application development," https://www.docker.com/, 2024, online; Accessed on 1 Dec. 2025.

[29] C. Denninnart, T. Chanikaphon, and M. Amini Salehi, "Efficiency in the serverless cloud paradigm: A survey on the reusing and approximation aspects," *Software: Practice and Experience*, vol. 53, no. 10, pp. 1853–1886, 2023.

[30] T. Chanikaphon and M. A. Salehi, "Ums: Live migration of containerized services across autonomous computing systems," in *GLOBECOM 2023-2023 IEEE Global Communications Conference*. IEEE, 2023, pp. 467–472.

[31] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[32] Shopify, "Toxiproxy: A tcp proxy to simulate network and system conditions for chaos and resiliency testing," https://github.com/Shopify/toxiproxy, 2024, online; Accessed on 1 Dec. 2025.