Case Study: How Network Bridges Outperform IoT Connectors in Industrial Applications on Edge

Shubhadeep Bhattacharya
opro.ai
shubhadeep.bhattacharya@opro.ai

September 25, 2025

Abstract

This case study investigates the deployment of MQTT in industrial environments, comparing standalone MQTT brokers with protocol handlers and proxy-based approaches. The study emphasizes key dimensions such as resilience, interoperability, performance tuning, application availability, consistency. Through empirical observations across multiple sites, we analyze the trade-offs between lightweight standalone brokers and more complex streaming integrations, highlighting where each approach provides operational advantages and where limitations become evident.

Contents

1 Introduction

IoT applications are increasingly deployed at the edge, often in constrained environments where compute power is limited, networks introduce latency, and system management expertise day-2 operations is scarce. At the same time, industrial IoT adoption has accelerated, with analytics, machine learning, and AI workloads relying on reliable data movement from sensors to edge and cloud platforms. For industries where downtime directly impacts safety and productivity, ensuring resilient and available data flows is critical and so is consistency to give operators the trust to adopt modern data based industrial applications.

In this scenario MQTT has emerged as a lightweight messaging protocol, but its deployment model can make or break operations whether as a standalone broker or through protocol handlers and proxy-based integrations that introduce trade-offs. This study examines the challenges of adopting protocol handlers and connectors, both open-source and commercial, and contrasts them with standalone MQTT deployments. We highlight the strengths and limitations of each approach and outline strategies to address operational challenges in industrial IoT systems.

2 Background

When industrial applications seek to employ artificial intelligence, advanced analytics, or machine learning as part of their operations, it often becomes evident that the interfaces required to connect their assets are not compatible with the modern data stack.Seamless operational data from diverse sites including PLCs, PACs, flow computers, OPC servers, machinery, and sensors is fundamental to data-driven solutions.

Both modern and legacy systems communicate through a wide range of protocols, including standardized frameworks such as OPC UA, OPC HA, and OPC DA, as well as vendor-specific implementations such as Siemens MPI and Allen Bradley DFI. These protocols are not inherently compatible with contemporary event streaming platforms, which are designed to ingest data from tens to hundreds of sensors in real time, in ordered sequences. This incompatibility necessitates a protocol translation layer, commonly referred to as an IoT gateway, which functions as a bridge between industrial sensor networks and the broader internet.The IoT gateway typically represents the first network hop between a plant sensor network and external systems. At this stage, sensor data is aggregated, transformed, timestamped, and subsequently forwarded to the internet or backend systems for advanced processing and analysis. Because the IoT gateway often resides in a network-constrained environment, commonly situated behind corporate firewalls or within air-gapped configurations, it becomes critical to ensure that backend systems can maintain a seamless, uninterrupted, and secure connection with the IoT gateway. This study contrasts network bridges against adapters or protocol handlers using an example of a real deployment in an Oil and Gas site. Some overlapping work exists, such as the evaluation of MQTT bridge architectures in cross-organizational contexts [1], which benchmarks MQTT bridges between brokers separated by networks across organizational boundaries. However, our work is unique in that it evaluates protocol handlers and proxies (used interchangeably here) versus bridges *within the same organization*, focusing on a full-scale MQTT deployment relaying messages to an event streaming platform. It is important to emphasize that this work is not aimed at comparing various MQTT broker implementations, as explored in [2], nor at analyzing general communication protocols in federated learning environments as studied in [3]. Instead, this study specifically focuses on the architectural trade-offs between standalone MQTT deployments and protocol handler or proxy-based approaches in industrial IoT

environments.

2.1 IoT Gateway

The system hosting the IoT gateway, usually located on site, is responsible for two primary tasks before publishing real time data:

- **Hosting an OPC server or similar system** that can ingest real time operational data from various devices. Typically, this involves interfacing with devices using the OPC protocol. However, there are also proprietary systems that communicate with vendor specific devices. This forms the ingress to the system running the IoT gateway.
- **Publishing real time data from the IoT gateway** using various protocols. Some of the widely adopted protocols include MQTT, REST, ThingWorx, CoAP (Constrained Application Protocol), and AMQP (Advanced Message Queue Protocol). This forms the egress from the system running the IoT gateway.

The table below highlights some of the strengths and weaknesses of each protocol. Depending on the use case, expertise in maintenance, application requirements, and the compute environment, one protocol may be more suitable than another.

Protocol	**Best Use Case**	**Strengths**	**Weaknesses**
MQTT	Lightweight telemetry, cloud IoT	Low bandwidth, QoS, broad adoption	Limited security; not suitable for high throughput
CoAP	Constrained devices, lossy networks	Lightweight, UDP based, supports multicast	Reliability challenges; small ecosystem
AMQP	Enterprise, financial IoT	Strong reliability and routing features	Heavy protocol; resource intensive
OPC UA	Industrial automation	Rich data modeling; strong security; interoperability	Complex to configure; high overhead
DDS	Real time robotics and vehicles	Low latency; peer to peer; fine grained QoS	Complex to deploy; heavier footprint
HTTP/HTTPS	Simple IoT web integration	Ubiquitous; secure via TLS	Verbose; not suitable for real-time.

Table 1: Comparison of IoT Communication Protocols

This case study examines a scenario in which a backend platform is deployed within an on-premises, air-gapped environment. In this setting, the streaming platform ingests messages from a sensor network to enable real-time inference and optimize the performance of industrial assets. The consistency of critical process tags, as well as the availability of the backend during active engagement with industrial equipment, are of paramount importance. While batch processes in the oil and gas sector generally tolerate relaxed latency requirements and there are examples of ultra-optimized ML inference (e.g. using FPGA or highly specialized hardware

achieving 1-3 ms latency in critical control settings [5]), most machine learning-based forecasting, optimization, and meta-heuristic approaches (such as genetic algorithms) operate with latencies in tens to hundreds of milliseconds or more. Studies on edge object detection [7] and inference latency prediction [6] reveal that communication, model complexity, and hardware constraints typically prevent sub-second general-purpose forecasting / optimization from being used in tight control loops.

It is important to note that requirements and outcomes differ significantly in hybrid on-premises and cloud environments. However, because industrial applications typically mandate that data remain on site [4], hybrid approaches are outside the scope of this study.

2.2 Network Bridges

Act as protocol translators and traffic optimizers. Support heterogeneous industrial systems (MQTT, Kafka, OPC-UA, etc.).For the purposes of this case study, the focus is on MQTT brokers, given their lightweight nature and widespread adoption in industrial settings. A network bridge entails deploying a fully featured MQTT broker that ingests messages from IoT gateways located at industrial sites and forwards them to the backend streaming platform as illustrated below.

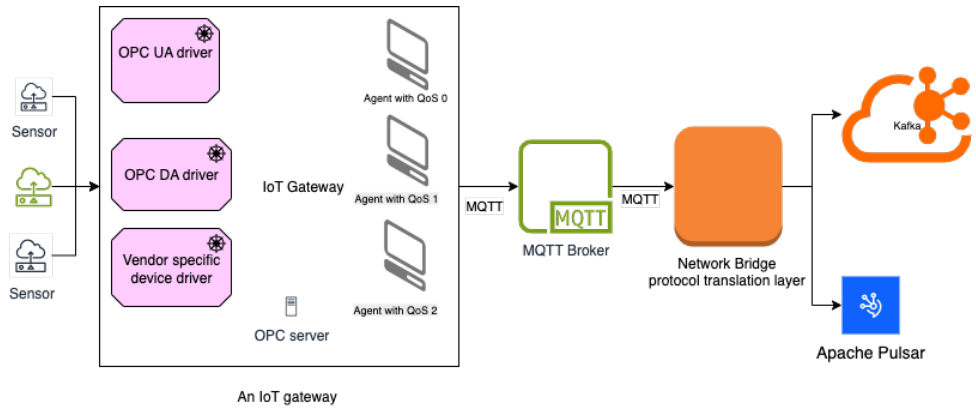

Figure 1: System architecture of a network bridge

2.3 Protocol Adapters, Proxies, and Handlers

Typically limited to point-to-point transformations.Alternative approaches may be adopted when optimizing for a smaller compute footprint. These include protocol adapters, connectors, or proxies. Common examples are MoP (MQTT on Apache Pulsar), MQTT proxies, or solutions such as Zilla Multi-Proxy Edge, which allow a single Kafka or Apache Pulsar deployment to natively support both MQTT protocol traffic and backend streaming workloads.

Enterprise-grade offerings, such as Confluent's managed Kafka, are excluded from this discussion due to their prohibitive cost and, more importantly, regulatory constraints in critical industrial sectors that restrict the transmission of operational data to cloud environments.

In these alternatives, the protocol handler enables MQTT to run natively on platforms such as Pulsar, Kafka, and other data-streaming systems, thereby reducing the need for a standalone broker layer.

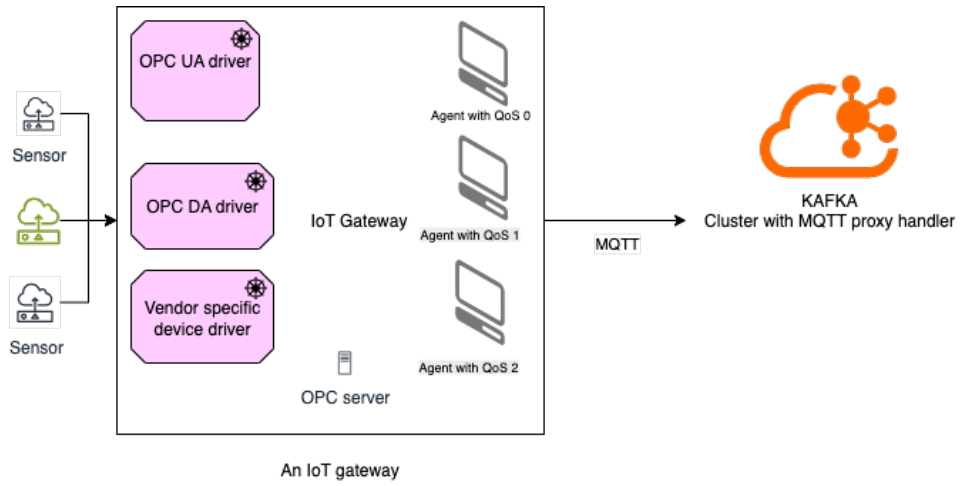

Figure 2: System architecture of a protocol handler or MQTT proxy

3 Deep Dive: Proxies and Connectors in industrial deployments

Across multiple deployments in the oil and gas and cement industries, particularly in on-premises, air-gapped environments where IoT gateways feed into backend platforms, several recurring challenges have been observed when using protocol handlers, proxies, or connectors.

3.1 Availability

Protocol handlers and proxies typically run within the same process as the backend streaming platform. Consequently, any downtime in the backend results in immediate loss of incoming telemetry, operational, or event data unless an alternative buffering mechanism exists. This undermines the reliability expected of mission-critical systems. Furthermore, most industrial sites lack the operational expertise to manage complex recovery workflows, requiring solutions with minimal operational overhead.

3.2 Connection Stability and Message Throughput

MQTT and Kafka/Pulsar have fundamentally different networking and system requirements, creating challenges in production-grade environments:

- **MQTT**: Resilient to disruptive networks and client unavailability, and capable of recovering session state after disconnections. For operational data, QoS 0 (“fire-and-forget”) is often sufficient, while critical process tags require QoS 1 or higher. However, frequent connection drops result in unacknowledged QoS 1+ messages, leading to unreliability or sluggish performance.
- **Kafka/Pulsar/MQ**: Require stable network connections. If downstream consumers operate on constrained compute resources, message delivery often fails, introducing operational risk.

3.3 Incomplete MQTT Protocol Support

Protocol handler projects remain in varying stages of maturity and do not yet provide robust, full-featured MQTT support. This creates gaps for industrial applications where engineers rely on:

- **LWT (Last Will and Testament)** to signal device health.
- **QoS semantics**: Full support for QoS 0, 1, and 2 is lacking. For instance, MoP (MQTT on Pulsar) does not support QoS 2, and Kafka's stream-first architecture is ill-suited for ordered queues required for deterministic actions (e.g., OSISoft PI Analytics). Lack of guaranteed ordering can cause incorrect or failed workflows.

3.4 Limited Visibility and Management

Protocol handlers often treat MQTT as a secondary protocol, resulting in limited visibility and administrative control. Observed shortcomings include:

- Basic SSL implementations with limited enterprise-grade options.
- Insufficient RBAC granularity (e.g., per-topic filters for securing critical tags).
- Limited monitoring capabilities: while Prometheus metrics can be enabled, site operations teams often lack expertise to interpret them. In contrast, standalone MQTT brokers typically provide intuitive dashboards with visibility into system state, network health, throughput, and client sessions.

3.5 Rigid Type Safety

Modern streaming platforms often enforce strict type safety through schema registries (e.g., Pulsar Schema Registry). While useful in many contexts, this rigidity conflicts with industrial IoT realities. Sensor data may originate from diverse OPC servers, and gateway OPC drivers may change due to security or operational requirements. This can cause schema mismatches, as protocol handlers typically lack mechanisms for on-the-fly schema inference.

3.6 Consistency Challenges

Partial data loss is a recurring concern. Critical asset events—such as temperature breaches or state changes—may be missed when protocol handler processes or parts of the streaming architecture are unavailable. Without QoS 2 (exactly-once) guarantees or local persistence, replaying events reliably becomes difficult. This results in a "split-brain" condition between the IoT gateway and the backend system, where each assumes a different state, undermining the consistency required for safe and efficient industrial operations.

3.7 Advantages of Using a Protocol Handler

While protocol handlers present several shortcomings in industrial contexts, there are also advantages to their use. A summary of these advantages is provided below:

- **No Additional Setup:** Although MQTT brokers are lightweight in terms of memory and compute footprint, deploying them introduces additional infrastructure components that must be maintained, deployed, and secured. Protocol handlers eliminate this requirement by integrating MQTT support directly within the streaming platform.
- **Lower Latency:** Protocol handlers enable IoT gateways to communicate directly with the platform's data layer, avoiding an additional network hop. This can be particularly valuable in industrial processes where latency is critical, such as in steel production or cooling systems in nuclear power plants, where response times are essential for safety and efficiency.
- **Unified Monitoring:** Once workflows mature and stabilize, monitoring the streaming infrastructure alone is generally sufficient. However, in environments where sensors evolve frequently and multiple MQTT clients connect and disconnect, centralized monitoring through a protocol handler simplifies operational oversight by consolidating observability into a single layer.

4 Methodology

4.1 Hardware

The experimental setup consisted of a modest server cluster of three nodes, equivalent to `m5.xlarge`. The cluster was provisioned using Rancher with RKE2 server (`v1.26.13+rke2r1`) running CentOS Stream. The cluster comprised one control-plane node and two worker nodes, supporting machine learning workloads.

4.2 Software

4.2.1 MQTT

Multiple OPC servers fed sensor data into KepwareEX v6.12 (a trademark of PTC Inc.). Using the IoT Gateway, telemetry, operations, and event data were streamed to the cluster via MQTT. HiveMQ Enterprise (self-hosted, running MQTT v3.1.1) was evaluated in both standalone and clustered modes. The standalone deployment was found to be highly resilient for industrial applications where sensor counts remained in the low hundreds and did not exceed 700 per asset per site. The broker demonstrated resilience to compute spikes, and MQTT clients were able to resume sessions without breaching service-level agreements (SLAs).

4.2.2 Apache Pulsar

Apache Pulsar (version 2.10.5 and above) was deployed in both standalone and clustered configurations to consume backend events. Additionally, the MQTT-on-Pulsar connector was evaluated under both deployment modes.

4.2.3 Kafka

The Confluent Platform 7.2 (with Apache Kafka 3.2) was evaluated in both standalone and clustered modes. An MQTT proxy for the Confluent Platform (supporting MQTT v3.1.1) was also deployed to production.

4.2.4 Network Bridge

A lightweight and stateless network bridge was evaluated, designed to consume messages from the MQTT broker and publish them to either Apache Pulsar or Apache Kafka.

4.3 Workload

The cluster was deployed on an oil and gas site, receiving data from four fractionation processes via MQTT.

MQTT messages were categorized into three groups based on QoS levels:

- **QoS 0:** Operational data from sensors were transmitted using "fire-and-forget" semantics with no acknowledgment, representing approximately 85% of all data.
- **QoS 1:** Operator limits, engineering limits, and constraints were sent with delivery acknowledgment to ensure backend systems reflected operational realities. This accounted for roughly 14% of total data.

- **QoS 2:** On/off switches and other critical event tags required exactly-once delivery to guarantee backend consistency with real-world states. This comprised approximately 1% of total data.

Messages arriving via OPC were published to the backend by KepwareEX MQTT agents using the following schema:

```
{
  "value": "25.3C",
  "quality": "good",
  "datetime": "2025-09-20T14:30:00Z"
}
```

4.4 Throughput Analysis

The theoretical maximum throughput was calculated assuming event-based tags with QoS 2 and an update frequency of 1 second per sensor:

- Fixed header: 2 bytes
- Variable header: 15 bytes
- Payload: 66 bytes
- Total per sensor message: 83 bytes

For 5,600 sensors:

$$5600 \times 83\,bytes = 464.8\,KB/s$$

This corresponds to approximately 40 GB of data per day. In practice, observed data volumes were 15–30% lower, depending on site availability, demand, and sensor activity.

5 Evaluation

5.1 Challenges

5.1.1 Pulsar

Cluster Mode: Pulsar cluster mode was operations-heavy due to the number of components that had to be managed:

- **ZooKeeper** — in the control plane, managing cluster metadata.
- **BookKeeper** — in the data plane, managing storage.
- **Brokers**.

Standalone Mode: The standalone mode initially scaled well, utilizing multiple CPU cores efficiently. The bottleneck during scaling was memory, which, although not prohibitively expensive, became a constraint. As the number of assets increased, higher partitioning was required, necessitating a transition to cluster mode.

Cluster Mode: The cluster mode was significantly more operations-heavy. Brokers occasionally entered states from which they would not recover, often due to incompatibilities between BookKeeper and the underlying storage class. This led to data loss, split-brain problems, and application unavailability.

5.1.2 Kafka

Cluster Mode: Kafka exhibited non-linear scaling challenges due to its monolithic broker design, where storage and compute scale together. This design is inefficient in Industrial IoT environments, where server sizes are difficult to scale. Misconfiguration or disk pressure often caused sluggishness. Although performance improved with upgraded server specifications, sluggishness persisted during peak throughput. A substantial amount of time was spent tuning the Kafka cluster, making self-hosting particularly challenging for industrial applications.

5.2 Evaluation Parameters

The setup was evaluated using the following hardware configuration:

- `m5.xlarge`: 3 nodes, 16 vCPUs each, 64 GiB memory.

5.3 Latency

Average end-to-end latency was measured at a throughput of 464.8 Kb/s with 5600 transactions per second (TPS).

QoS Level	Network Bridge	Pulsar (MQTT on Pulsar)	Kafka (MQTT Proxy)
QoS 0	9.09 ms	7.89 ms	6.82 ms
QoS 1	15.83 ms	12.97 ms	13.58 ms
QoS 2	27.56 ms	N/A	21.32 ms

Table 2: Average latency across different QoS levels.

5.4 Resource Utilization

5.4.1 Memory

Average memory usage was measured under the same throughput and TPS for a period of 15 minutes.

QoS Level	Network Bridge	Pulsar (MQTT on Pulsar)	Kafka (MQTT Proxy)
QoS 0	56%	53%	48%
QoS 1	62%	68%	56%
QoS 2	68%	N/A	83%

Table 3: Average memory utilization across QoS levels.

Less variability and fewer memory spikes were observed when using a network bridge, even when Kafka was paired with a full-scale Hive MQTT broker.

5.4.2 Compute

Average CPU utilization was measured under the same throughput and TPS for a period of 15 minutes.

Kafka's MQTT proxy exhibited over-utilization when QoS levels greater than 1 were required. Accounting for this variability in CPU usage is particularly challenging in self-hosted environments, where resource scaling requires careful planning.

QoS Level	Network Bridge	Pulsar (MQTT on Pulsar)	Kafka (MQTT Proxy)
QoS 0	68%	73%	48%
QoS 1	73%	82%	53%
QoS 2	81%	N/A	94%

Table 4: Average CPU utilization across QoS levels.

5.5 Consistency

The most significant challenge when using connectors and proxies in a self-hosted environment was achieving consistency. Kafka's monolithic architecture made it tedious to ensure that both the MQTT service and the Kafka broker service maintained sufficient uptime. Designing strategies to handle event-based QoS level 2 tags when MQTT was unavailable proved difficult. By contrast, employing a network bridge provided flexibility by persisting data locally and/or informing on-site operations using Last Will and Testament (LWT) messages when the Kafka or Pulsar backend was unavailable.

6 Results

After deploying across 6+ sites spanning multiple sectors, with Operations (Ops) teams ranging in size from 5 to 20, the following results were observed.

6.1 Standalone MQTT

Resilience: A standalone MQTT server rarely fails due to memory or disk pressure. Clients and brokers recover effectively after a short interval (ranging from a few seconds to a few minutes) and are able to restore their session state. Resource spikes in memory and compute utilization are uncommon, even during connection surges or at peak message throughput.

Availability: Standalone MQTT demonstrates higher availability compared to protocol handler–based solutions. Given the premium on compute capacity at the edge, standalone MQTT offers an extremely low compute footprint. It remains resilient in heterogeneous clusters comprising diverse nodes, storage types, operating systems, and architectures (ranging from microprocessors to microcontrollers).

Consistency: Observed behavior indicates predictable session persistence and delivery guarantees under typical workloads, though formal verification under extreme network partition scenarios remains limited.

6.2 Comparative Analysis: Confluent MQTT-on-Kafka vs. MQTT on Pulsar (MoP)

The following table summarizes the comparative findings between two protocol handler approaches:

Feature / Dimension	Confluent MQTT-on-Kafka	MoP (MQTT on Pulsar)
Network / Scaling / Connections	Scales well due to lighter architecture (direct MQTT to kafka). Known issues observed under high load (crashes, disconnects).	Open issues with connection scaling, stability, and proxy connection drops.
Latency	Lower-latency path: MQTT proxy directly into Kafka.	Extra hops (proxy + Pulsar internals + network). Latency spikes observed under load.
Compute / Resource Use	CPU/memory usage proportional to MQTT connections plus Kafka I/O. Overhead with QoS2, session persistence, retain, etc.	Resource load from Pulsar Brokers + proxy + protocol handler. Reported resource leaks.
Session State / Persistence	Claimed support. Sessions stored for reconnect. Documented cases of session takeover; project is maturing.	Partial support. Reported problems: lost sessions, forced client disconnects, unsubscribed topics behaving incorrectly.
QoS (0,1,2)	Supports all three levels; issues noted with QoS 2.	Explicit open issue with QoS 2.
LWT (Last Will & Testament)	Supported.	Reported issues; LWT is not robust.
Delivery Semantics / Guarantees	Some risks under heavy load (retain + QoS2 + crash or partition). Issues with empty retained messages.	Multiple reports of message loss/drops during disconnects, proxy failures, or insufficient retry/reconnect logic.
Retained Messages	Both implementations degrade under extreme load.	Same limitations.
Shared Subscriptions / Wildcards / Topic Filtering	Supports wildcard topics (as per MQTT spec), with caveats under heavy load.	Issues with wildcard filtering.
Fault Tolerance / Network Partition / Broker Failures	Sessions, QoS, or retain messages may misbehave under Kafka cluster issues. Not fully validated across all failure modes.	Proxy-broker link failures lead to dropped connections. Permissions/auth failures can also cause disconnects. Weaknesses exposed in fault scenarios.

Table 5: Comparison: Confluent MQTT-on-Kafka vs. MQTT on Pulsar (MoP)

7 Conclusion

In hybrid environments or cloud , proxy handlers may offer value since elastic scaling and managed services would address most of the issues; however, in constrained on-premises settings, their limitations became evident. Even modest approaches such as introducing custom logic between the MQTT broker and the streaming platform—can mitigate many trade-offs despite latency been introduced due to extra hops.

Specifically, custom bridge logic enabled:

- Improved consistency and robustness.
- Support for diverse data schemas.
- Triggering tasks when QoS > 1 events arrive while the streaming platform is unavailable.
- Better resource utilization during spikes in throughput across QoS states.
- Event data storage and replay to rebuild the last known state.

Moreover, custom bridges provide flexibility to adapt to site-specific requirements, which can vary significantly across deployments. This approach offers a practical balance between operational constraints and reliability at scale.

References

[1] K. Lima, T. D. Oyetoyan, R. Heldal, and W. Hasselbring, "Evaluation of MQTT Bridge Architectures in a Cross-Organizational Context," *arXiv preprint arXiv:2501.14890*, 2025. Available: `https://arxiv.org/abs/2501.14890`

[2] M. Bender, E. Kirdan, M.-O. Pahl, and G. Carle, "Open-Source MQTT Evaluation," in *Proc. IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pp. 1–4, 2021. doi: 10.1109/CCNC49032.2021.9369499.

[3] G. Cleland, D. Wu, R. Ullah, and B. Varghese, "FedComm: Understanding Communication Protocols for Edge-based Federated Learning," *arXiv preprint arXiv:2208.08764*, 2022. Available: `https://arxiv.org/abs/2208.08764`

[4] Elias Dritsas and Maria Trigka, "A Survey on the Applications of Cloud Computing in the Industrial Internet of Things," *Big Data and Cognitive Computing*, vol. 9, no. 2, Article 44, 2025. DOI: 10.3390/bdcc9020044. Available: `https://www.mdpi.com/2504-2289/9/2/44`

[5] R. Shi, S. Ogrenci, J. M. Arnold, et al., "ML-based Real-Time Control at the Edge: An Approach Using hls4ml," *arXiv preprint arXiv:2311.05716*, 2023. :contentReference[oaicite:3]index=3

[6] Zhuojin Li, Marco Paolieri, Leana Golubchik, "Inference Latency Prediction at the Edge," *arXiv preprint arXiv:2210.02620*, 2022. :contentReference[oaicite:4]index=4

[7] Authors, "Inference Latency Prediction Approaches Using Statistical Information for Object Detection in Edge Computing," *Applied Sciences*, vol. 13, no. 16, Article 9222, 2023. :contentReference[oaicite:5]index=5