

AN EMPIRICAL STUDY OF PYTHON FOR SOFT REAL-TIME IOT STREAM PROCESSING: INVESTIGATING LATENCY, MQTT QOS/TLS, AND BATCHING AT THE EDGE

Анотація

У багатьох периферійних додатках Інтернету речей (IoT) для забезпечення безпеки та експлуатаційних вимог необхідна м'яка реакція в режимі реального часу. Хоча Python є основною мовою для створення прототипів у сфері IoT, його продуктивність та придатність як мови сценаріїв для легких систем обробки потоків не були ретельно досліджені й залишаються майже невивченими, часто часто відкидаючись через обмеження глобального блокування інтерпретатора (GIL). У цій статті представлено емпіричну оцінку синхронного однопотокового (S1) конвеєра Python з використанням відтворюваної тестової платформи за різних робочих навантажень, конфігурацій MQTT (QoS = 0/1/2) та шифрування TLS. Результати показують, що конвеєр S1 досягає затримки p99 менше мілісекунди при 10 Гц із кількістю видавців $N = 50$ та розміром корисного навантаження 512 Б, забезпечуючи відсутність пропущених дедлайнів за умов м'якого обмеження у 200 мс. Цікаво, що MQTT із рівнем QoS 2 демонструє меншу затримку, ніж QoS 0, коли видавець і підписник знаходяться на локальному хості. Крім того, застосування політики вікон із подвійним порогом ($\text{max_items} = 32$, $\text{max_ms} = 50$) забезпечує продуктивність менше мілісекунди під час тимчасового агрегування. В цілому, ці результати з легкістю відповідають м'якому дедлайну 200 мс, що вказує на значний запас і потенціал для виконання більш суворих вимог (наприклад, менше 100 мс) за подібних умов.

Ключові слова: IoT, системи реального часу, обробка потоків, Python, MQTT, периферійні обчислення, характеристика продуктивності

Introduction

With the explosion in the number of IoT applications, ultra-low latency has become imperative. For many such systems, data processing must occur within strict time constraints to ensure safety and operational performance. Edge gateway processing is an effective strategy to reduce network overhead and accelerate responses. However, the performance characteristics of lightweight, single-node

stream processors implemented in high-level languages such as Python are not yet well studied.

Python, with its rich ecosystem and rapid development capabilities [1], has become a popular choice for IoT prototyping and conceptualisation. However, the Global Interpreter Lock (GIL) serialises all thread execution, raising concerns about its real-time capabilities [3]. Existing research has mainly focused on JVM-based or compiled domain-specific language (DSL) frameworks, leaving several important gaps:

- (1) the lack of empirical Python latency and deadline-adherence baselines under typical IoT workloads [1];

- (2) an incomplete characterisation of the MQTT protocol and TLS encryption overhead [4];

- (3) the absence of systematic rules for selecting batching parameters [2]; and

- (4) few studies providing a reproducible testbed for such analyses [8].

This study addresses the following research questions:

- (1) Under different loads, what baseline latency (p50, p95, p99) and jitter can a synchronous Python pipeline achieve?

- (2) What is the impact of MQTT QoS levels in combination with TLS encryption on tail latency and deadline-miss rate?

- (3) What is the performance of dual-threshold windowing for message batching?

The main contributions of this paper are:

- (1) a reproducible and comprehensive baseline demonstrating that Python can achieve, to the best of our knowledge, sub-millisecond p99 latency for soft real-time IoT edge processing;

- (2) measurements of the overhead introduced by MQTT and TLS, including an anomaly observed with localhost QoS 2;

- (3) a characterisation of baseline windowing performance; and

- (4) an open-source description of a portable testbed [8].

Methodology

A synchronous, single-threaded (S1) pipeline was employed for deterministic processing of IoT sensor streams, establishing a baseline for real-time performance, reducing concurrency-induced variance, and decoupling protocol and batching effects. This design also avoids contention with the Global Interpreter Lock (GIL), which arises only under multi-threaded execution [3, 6]. A factorial experimental design was used to examine performance across stream operators and system configurations. The setting comprised publishers generating time-stamped messages with traffic patterns (periodic, on-off, Poisson) and payload sizes of 128–2048 B; an MQTT broker (Mosquitto 2.0.22) providing QoS

levels 0, 1, and 2 with optional TLS; and a single-threaded CPython 3.12 processor executing the operator chain. Per-message latency was measured using `time.perf_counter()`, and CPU usage, memory consumption, and throughput were recorded.

To perform the experimental setup, experiments were run on an Apple M4 Max (macOS) with Python 3.12.7. All components were run on localhost to rule out network-related variabilities and focus only on reporting System-Under-Test performance metrics. The evaluation included a factorial parameter space of: publisher count $N = \{1, 5, 10, 30, 50\}$, payload size (128 B / 512 B / 2048 B), traffic patterns (periodic, on-off, Poisson), MQTT QoS level (0, 1, 2), TLS encryption (enabled, disabled), and windowing configuration parameters $\text{max_items} = 32$, $\text{max_ms} = 50$. Three independent experiments were performed for each parameter, consisting of a 5-second warm-up, a 60-second measurement, and a 2-second cool-down.

Evaluation metrics were chosen to characterise system performance fully. Latency is assessed with percentile values (p50, p95, p99) from the sorted latency distribution. Jitter is defined as the standard deviation of latency measurements. The deadline miss rate is calculated as $\text{Miss Rate} = \frac{1}{N} \sum_{i=1}^N 1(L_i > D) \times 100\%$, where the deadline D is 200 ms. Throughput is measured as the number of messages processed per second. Resource utilisation includes CPU and memory usage. End-to-end latency is calculated as $L_{e2e} = T_{\text{receive}} - T_{\text{publish}}$, using monotonic timestamps to ensure accuracy.

Results and Discussion

RQ1: Baseline Performance

The S1 pipeline achieved consistent sub-millisecond p99 latencies across workload patterns ($N = 10$, 512 B):

Table 1.

Latency Performance Across Workload Patterns					
Pattern	p50 (ms)	p95 (ms)	p99 (ms)	Miss %	Throughput (msg/s)
Periodic	0.10	0.38	0.97	0.00	96.7
Poisson	0.09	0.35	0.89	0.00	96.5
On-off	0.11	0.39	0.98	0.00	96.4

All configurations maintained a 0.00% deadline miss rate against the 200 ms soft deadline. Throughput scaled nearly linearly ($R^2 \approx 0.99$) with publisher count:

Table 2.

Scalability with Publisher Count			
N	p99 (ms)	Throughput (msg/s)	CPU %
1	1.15	8.5	0.33
10	0.94	92.1	1.24
50	0.76	459.4	3.83

As the number of publishers (N) increased from 1 to 30, a surprising drop in tail latencies was observed. This could be attributed to better utilisation of CPU cache and the amortisation of per-message overhead [6]. As a result, the system sustained 459 messages per second at N = 50, with CPU utilisation remaining well below 4%, indicating substantial remaining capacity. The synchronous design eliminates context switching and thread contention, thereby preserving deterministic execution. In this single-threaded architecture, the Global Interpreter Lock (GIL), a well-known Python limitation frequently discussed in the literature, is not a bottleneck [3, 6].

RQ2: Protocol Effects (QoS Anomaly)

MQTT QoS levels and TLS encryption exhibited unexpected latency patterns (localhost, 512 B):

Table 3.

QoS and TLS Impact on Latency				
QoS	TLS	p99 (ms)	Δ vs no-TLS	CPU %
0	False	1.04	—	1.69
0	True	0.98	-6%	2.12
1	False	1.09	—	2.15
1	True	0.80	-27%	1.48
2	False	0.44	—	1.28
2	True	0.92	+109%	2.21

QoS 2 without TLS achieved the lowest latency (p99 = 0.44 ms), which is 58% lower than QoS 0 (p99 = 1.04 ms). This result is surprising, since QoS 2's four-way handshake would typically be expected to add latency [4]. This outcome is a localhost-only artefact, caused by Mosquitto broker optimisations that pipeline or coalesce the handshake when network delay is negligible [5]. Such behaviour is unlikely to generalise to WAN deployments, where the expected latency order (QoS 0 < QoS 1 < QoS 2) is expected to hold [7].

Adding TLS reduced latency by 6% and 27% for QoS 0 and QoS 1, respectively. This improvement could be attributed to hardware-accelerated cryptographic primitives (e.g., Apple/ARM crypto accelerators) that benefit from good cache locality. In contrast, for QoS 2, latency increased by up to 109% when TLS was enabled, indicating that the overhead of TLS processing compounded with the ACK sequence overhead inherent to QoS 2 [7].

RQ3: Windowing Performance

The baseline dual-threshold configuration ($\text{max_items} = 32$, $\text{max_ms} = 50$) consistently maintained sub-millisecond performance:

Table 4.

Windowing Parameter Performance				
max_items	max_ms	p99 (ms)	Miss (%)	Throughput (msg/s)
32	50	0.96	0.00	175.19

This mechanism enables adaptive system behaviour. Under high load, the max_items parameter primarily determines throughput, whereas under low load, max_ms prevents unbounded queuing. The latency-throughput trade-off can be expressed as $L_{\text{batch}} \approx \frac{W}{2} + \frac{C(B)}{B}$, where W is the waiting time and $C(B)$ is the per-batch processing cost. For stricter latency constraints (< 50 ms), set max_items to ≤ 16 and max_ms to ≤ 10 . For workloads prioritising throughput (> 200 ms), we recommend using $\text{max_items} = 64$ and $\text{max_ms} = 50$ [2].

Conclusion

The presented empirical results confirm the feasibility of synchronous Python pipelines for soft real-time IoT edge applications. With a p99 latency of 0.96 ms and zero deadline misses, the S1 pipeline challenges common misconceptions about Python’s performance in time-sensitive systems. For localhost deployments, MQTT QoS 2 produced remarkably low latencies (p99 = 0.44 ms), an artefact of localised broker optimisations; however, these results do not generalise to wide-area networks [5]. Interestingly, for QoS 0 and QoS 1, TLS encryption reduced latency, whereas it increased QoS 2 response time by 109%, indicating that the effect is configuration-dependent [7]. The fixed dual-threshold windowing policy ($\text{max_items} = 32$, $\text{max_ms} = 50$) enabled the system to maintain sub-millisecond latency without disruptions from adaptive strategies [2].

Nevertheless, these results were obtained from single-node localhost setups ($n = 3$ trials per configuration) and therefore are not representative of real-world network conditions. In addition, the use of synthetic workloads may not fully capture the complexity and randomness of real IoT traffic. Although this sample

size is sufficient to draw baseline conclusions, the statistical power of future studies could be enhanced by employing bootstrap confidence intervals. Results were obtained on localhost with a single-threaded processor; external network latencies, different hardware, or security settings may narrow the apparent headroom.

Rooted in these observations and limitations, future work will evaluate the proposed approach on real edge devices (e.g., Raspberry Pi 4) under realistic network conditions, including Wi-Fi and cellular connections, with controlled latency injection via *netem* for WAN emulation. Additionally, asynchronous (asyncio-based) pipelines will be benchmarked to examine the trade-off between concurrency and latency. We also plan to investigate the integration of machine learning inference operators (e.g., ONNX and TensorFlow Lite) to characterise their impact on real-time performance guarantees in edge AI systems [6].

According to the results, synchronously and single-threaded implemented Python pipeline can deliver low tail latencies and does not miss a single deadline under the analysed setups with a soft 200 ms deadline. The observed headroom indicates potential for tighter budgets (e.g., sub-100 ms) in similar local conditions, but the degree of this headroom will depend heavily on the deployment environment and configuration.

References

1. S. Y. Cho, R. Delgado, and B. W. Choi, "Feasibility study for a Python-based embedded real-time control system," *Electronics*, vol. 12, no. 6, p. 1426, Mar. 2023, doi: 10.3390/electronics12061426.
2. T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. ACM Symp. Cloud Computing*, 2014, pp. 1-13, doi: 10.1145/2670979.2670995.
3. O. Castro, P. Bruneau, J. Sottet, and D. Torregrossa, "Landscape of high-performance Python to develop data science and machine learning applications," *ACM Comput. Surv.*, vol. 56, no. 2, pp. 1-39, Feb. 2024, doi: 10.1145/3617588.
4. F. Palmese, A. E. C. Redondi, and M. Cesana, "Adaptive quality of service control for MQTT-SN," *Sensors*, vol. 22, no. 22, p. 8852, Nov. 2022, doi: 10.3390/s22228852.
5. J. Dizdarević, M. Michalke, and A. Jukan, "Engineering and experimentally benchmarking open source MQTT broker implementations," arXiv preprint arXiv:2305.13893, 2023.
6. V. Stoico, A. C. Dragomir, and P. Lago, "An empirical study on the performance and energy usage of compiled Python code," arXiv preprint arXiv:2505.02346, 2025.

[7] T. Prantl et al., "Performance impact analysis of securing MQTT using TLS," in *Proc. ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, 2021, pp. 241-248, doi: 10.1145/3427921.3450253.

[8] M. Vučinić, B. Škrbić, E. Kočan, M. Pejanović-Djurišić, and T. Watteyne, "OpenBenchmark: Repeatable and reproducible Internet of Things experimentation on testbeds," in *Proc. IEEE INFOCOM Workshops*, 2019, pp. 289-294, doi: 10.1109/infcomw.2019.8845160.