ISSN 2616-5775 Vol. 8, Issue 11: 71-78, DOI: 10.25236/AJCIS.2025.081108

SlotFlow: In-Memory File-Slot Orchestration and Adaptive Backpressure Mechanism for Constrained Edge Environments

Jingyang Liu

Hunan University of Science and Technology, Xiangtan, Hunan, 411201, China

Abstract: *In edge computing environments, device resources are often highly restricted. To cope with increasingly complex deep learning tasks, decomposing monolithic models into multiple collaborative microservice containers has become an effective approach to break through computing power bottlenecks. However, existing container orchestration solutions (e.g., K3s, KubeEdge) suffer from excessive control plane overhead, redundant network protocol stacks, and a lack of fine-grained flow control on the edge. Furthermore, the NAND Flash storage media widely used in edge devices face serious risks of write amplification and wear. Addressing these challenges, this paper proposes SlotFlow, a lightweight orchestration architecture based on In-Memory File-Slots. The core idea of SlotFlow is to utilize the host's tmpfs as a zero-copy communication bus, achieving microsecond-level synchronization through atomic file operations. Additionally, it introduces a backpressure mechanism based on queueing theory and control theory to effectively prevent memory overflows. Experimental results demonstrate that SlotFlow's communication performance on a single node approaches that of Unix Domain Sockets, reducing latency by approximately 84% compared to traditional TCP loopback. Theoretically, it reduces physical Flash writes to zero. In overload scenarios, SlotFlow's backpressure mechanism reduces task queue backlog by approximately 73% and extends the system's crash-free runtime by 2.5 times compared to the control group, achieving highly robust dynamic task orchestration.*

Keywords: *Edge Computing; Container Orchestration; Inter-Process Communication (IPC); tmpfs; Backpressure Mechanism; Queueing Theory; Dynamic Routing; Microservice Orchestration; Zero-Copy Communication*

1. Introduction

With the deep integration of the Internet of Things (IoT) and Artificial Intelligence (AI), Edge Intelligence (Edge AI) is rapidly moving from concept to implementation. In scenarios such as autonomous driving, industrial quality inspection, and smart security, a single deep learning model often fails to meet complex business demands. Decomposing complex monolithic applications into multiple heterogeneous containers (e.g., one container for video decoding, one for object detection, and another for logical decision-making) for collaborative inference has gradually become a mainstream practice to improve resource utilization, as demonstrated by the multimodal Edge AI framework proposed by Pan et al. [1]. Research by Li et al. also indicates that fine-grained model partitioning can effectively enhance inference efficiency in heterogeneous edge networks [2].

However, directly migrating "Cloud Native" orchestration concepts to edge devices (such as Raspberry Pi, NVIDIA Jetson) faces serious "compatibility issues" (or context mismatch):

First, Orchestration Overhead and Communication Bottlenecks. Traditional Kubernetes (K8s) and its lightweight versions (e.g., K3s) rely on complex control plane components (API Server, etcd, Kubelet). Comparative analysis by Yakubov et al. points out that these components consume substantial CPU and memory resources on resource-constrained devices, leading to an inversion phenomenon where "the system consumes more resources than the application" [3]. Furthermore, studies by Alqaisi et al. [4] and Gupta et al. [5] quantify the performance loss caused by Container Network Interfaces (CNI), noting that communication mechanisms based on Bridge or Overlay networks introduce non-negligible serialization and kernel protocol stack overheads when handling high-frequency real-time tasks.

Second, Storage Lifespan and Reliability Crisis. Most edge devices use NAND Flash (e.g., SD cards or eMMC) as primary storage. These media have limited Program/Erase (P/E) cycles. Existing orchestration systems and application logs tend to perform frequent disk writes, leading to severe Write

Amplification effects. Oh et al., in their work on the MiDAS system, pointed out that log-structured file systems significantly accelerate hardware aging when processing small block writes [6], which may cause edge devices to become paralyzed due to storage failure during long-term unattended operation.

Third, Lack of Flow Control Mechanisms. In heterogeneous computing environments, the processing speeds of upstream and downstream tasks are often mismatched. The lack of effective backpressure mechanisms leads to indefinite data accumulation in buffers, eventually triggering Out of Memory (OOM) errors. Ekane et al., in the DiSC system, emphasized that without cross-layer backpressure feedback, excess load generated upstream can cause a collapse in overall system throughput [7]. Although data center networks have mature flow control algorithms, inter-container communication at the edge often lacks this application-layer adaptive protection.

To address these challenges, this paper proposes SlotFlow. Unlike traditional solutions dependent on network protocol stacks, SlotFlow returns to the UNIX design philosophy of "everything is a file," constructing a communication plane that resides entirely In-Memory. By combining the zero-physical I/O characteristics of tmpfs with a metadata-driven dynamic routing protocol, SlotFlow ensures microsecond-level communication latency while completely solving the Flash wear problem.

2. Related Work

2.1. Limitations of Edge Container Orchestration

Currently, lightweight K8s distributions represented by KubeEdge, MicroK8s, and K3s dominate the edge orchestration market. However, recent research by Yakubov and Hästbacka [3] points out that even after deep pruning, the idle memory footprint of K3s can still reach hundreds of megabytes, which is a heavy burden for edge devices with only 2GB or 4GB of memory. Alqaisi et al. [4] further analyzed the performance of computer vision applications in edge containers and found that the network virtualization layer is a major source of latency. Characterization analysis by Gupta et al. [5] also confirmed that the I/O overhead brought by containerization is particularly significant when handling small-packet, high-frequency communication. In contrast, SlotFlow abandons the general container network abstraction and focuses on high-performance in-memory collaboration within a single node, aiming to eliminate these unnecessary intermediate layer overheads.

2.2. Edge Deployment of Deep Learning Models

To run large models on the edge, researchers have proposed various strategies. Li et al. [2] explored fine-grained Model Partitioning techniques, dynamically allocating DNN layers to different computing units to optimize latency. Matsubara et al. [8] surveyed "Split Computing" and "Early Exiting" mechanisms, emphasizing that communication efficiency after task segmentation is the system bottleneck. SlotFlow is designed specifically to provide underlying communication support for such frequent interactions following fine-grained segmentation, ensuring that computational gains are not offset by communication overheads.

2.3. Flow Control and Backpressure

In distributed systems, backpressure is key to preventing overload. Ekane et al. [7], in the DiSC project, demonstrated the importance of propagating backpressure in multi-tier applications, noting that static queue management often fails under highly dynamic loads. Traditional Socket communication relies on the TCP sliding window for flow control, but in single-machine container scenarios, the TCP stack is too heavy and agnostic to application-layer semantics. SlotFlow draws on control theory ideas to design a lightweight backpressure mechanism based on shared memory atomic counters, filling this gap.

3. System Architecture

SlotFlow adopts a "Host-Container" two-layer monitoring model. The key to the architecture is placing all task slots (Slots), state files, and intermediate feature tensors entirely within the in-memory file system (tmpfs).

3.1. Physical Layer: Memory as the Bus

SlotFlow directly uses /dev/shm or a mounted tmpfs volume as a shared bus. Research by Oh et al. [6] indicates that eliminating unnecessary physical Flash writes is crucial for extending device life. To this end, SlotFlow mandates the use of the noswap option during mounting, ensuring data remains 100% resident in physical memory. This prevents the kernel from swapping pages to disk under memory pressure, thereby completely avoiding disk I/O bottlenecks and media wear. The overall system architecture of SlotFlow is shown in Figure 1, which displays the host's shared memory bus (tmpfs) and multiple containers communicating via File Slots, with the Control Layer (Slot Manager) responsible for monitoring slot status and dynamic routing.

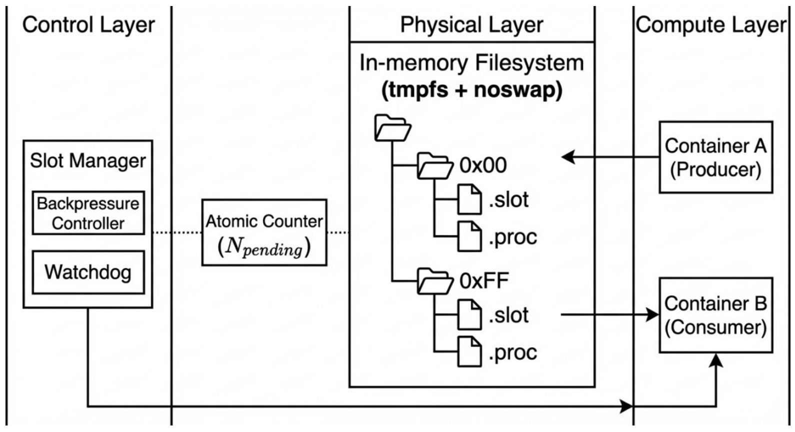

Figure 1: SlotFlow Overall System Architecture

3.2. Communication Protocol: Atomic File Slots

To ensure concurrency safety in a lock-free environment, we designed a three-phase protocol (Publish-Acquire-Commit) based on POSIX atomic file operations:

Publish: The upstream container first writes data into a .tmp temporary file, then atomically renames it to a .slot file using os.rename.

Acquire: The downstream container listens for directory events via inotify. Once a new .slot file is detected, it is atomically renamed to a .proc-{PID}-{timestamp} file. This is equivalent to acquiring a mutex lock with a lease.

Commit: After processing, the downstream container writes the result back to the file (or next stage), deletes the .proc file, and releases the slot.

The system supports dynamic adjustment of the number of task slots. The Slot Manager optimizes parallelism by adding or merging slot directories based on queue depth and container load. The routing strategy is implemented based on metadata files (e.g., route.json), which can direct .slot files to different consumer containers based on task type (e.g., image classification or object detection). The timing diagram of its communication protocol is shown in Figure 2, illustrating the entire process from publishing to commitment, as well as possible timeout rollback paths.

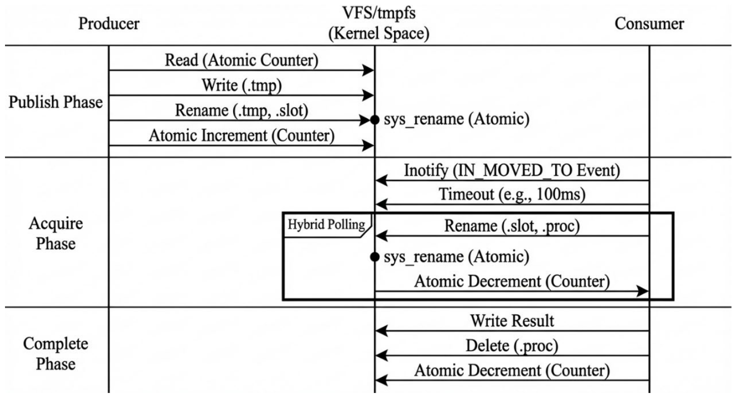

Figure 2: Communication Protocol Timing Diagram

3.3. Dynamic Routing Strategy Example

SlotFlow supports dynamic routing based on JSON metadata. The following is a typical routing configuration example:

JSON

json

```
{
   "task_type": "object_detection",
   "model_version": "yolov5s",
   "target_container": "detector_container",
   "priority": "high",
   "timeout_ms": 100,
   "fallback_target": "detector_backup"
}
```

The routing decision adopts a "Shortest Queue First" strategy: The Slot Manager monitors the task queue length of each consumer container in real-time and dynamically assigns new tasks to the container with the shortest queue, thereby achieving load balancing.

3.4. Dynamic Routing Strategy Example

Addressing the issue of "tasks getting stuck due to consumer failure," SlotFlow includes a built-in Watchdog (Janitor) based on file leases. The Slot Manager periodically scans all .proc files; if the modification time exceeds a threshold (e.g., 5 seconds), it determines that the container has zombie-died and rolls back the task or moves it to a dead-letter queue.

4. Theoretical Modeling and Analysis

To thoroughly demonstrate SlotFlow's performance advantages and stability, this section performs mathematical modeling from the dimensions of queueing theory and control theory.

4.1. Latency Analysis Based on Queueing Theory

We model the inference pipeline of an edge node as an $M/M/1$ queueing system. Assuming task arrival follows a Poisson distribution with parameter λ, and service time follows an exponential distribution with parameter μ. According to Little's Law, the average number of tasks in the system L and the average residence time W satisfy $L = \lambda W$.

In the traditional Socket communication model, the service time for a single interaction μ_{socket}^{-1} includes multiple kernel-mode copies, consistent with the high system call overhead observed by Al-Ali et al. [3]:

$$\frac{1}{\mu_{socket}} = t_{compute} + t_{copy\_user2kern} + t_{schedule} + t_{copy\_kern2user} \quad (1)$$

In the SlotFlow model, since data resides in shared memory, communication involves only pointer operations at the VFS layer. Its service time $\mu_{slotflow}^{-1}$ is:

$$\frac{1}{\mu_{slotflow}} = t_{compute} + t_{vfs\_rename} \quad (2)$$

Since $t_{vfs\_rename} \ll t_{copy}$, we obtain $\mu_{slotflow} > \mu_{socket}$. This theoretically explains why SlotFlow can significantly reduce queue waiting time.

4.2. NAND Flash Write Amplification and Lifespan Model

Write Amplification Factor (WAF) is a key factor affecting Flash lifespan. WAF is defined as

$WAF = \frac{V_{flash}}{V_{host}}$. Oh et al. [6] pointed out that in journaling file systems like ext4, to ensure metadata consistency, even writing 1KB of logs may trigger a 4KB (Page) or even larger physical block erasure, resulting in $WAF \gg 1$.

For SlotFlow, due to the use of tmpfs and disabled Swap, all file operations are completed in DRAM:

$$V_{flash}^{SlotFlow} \equiv 0 \Longrightarrow Life_{expectancy} \to \infty \tag{3}$$

This means that no matter how frequent the interactions between containers are, the wear on physical storage media is completely eliminated through architectural design.

4.3. Control Theory Interpretation of Backpressure Mechanism

To solve the overload problem proposed by Ekane et al. [7], we model SlotFlow's backpressure mechanism as a nonlinear feedback control loop.

Controlled Object: The task queue in shared memory.

State Variable: Currently backlogged task count $N_{pending}$ (maintained by an atomic counter).

Control Law: Adopting logic similar to a Hysteresis Comparator:

$$u(t) = \{\begin{matrix} 1\ (Allow), & if\ N_{pending} < T_{warn} \\ 0\ (Drop), & if\ N_{pending} \geq T_{critical} \end{matrix} \tag{4}$$

The flow control and backpressure mechanism logic is shown in Figure 3, illustrating the closed-loop control process from queue monitoring to decision feedback.

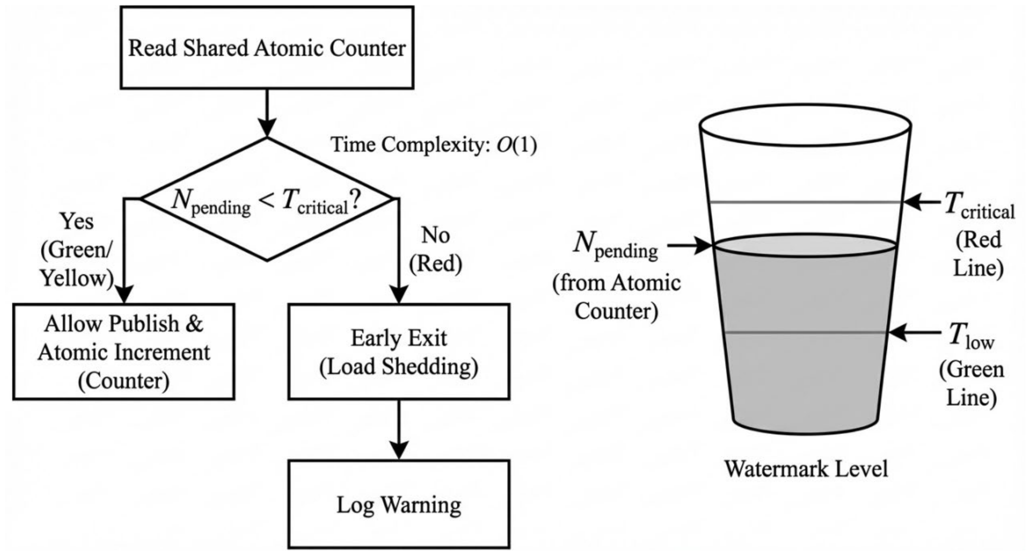
Read Shared Atomic Counter
Time Complexity: $O(1)$
$N_{\text{pending}} < T_{\text{critical}}$?
Yes (Green/ Yellow)
No (Red)
Allow Publish & Atomic Increment (Counter)
Early Exit (Load Shedding)
Log Warning
N_{pending} (from Atomic Counter)
T_{critical} (Red Line)
T_{low} (Green Line)
Watermark Level

Figure 3: Flow Control and Backpressure Mechanism Logic Diagram

Compared to traditional TCP window-based flow control, SlotFlow leverages global visibility in a single-machine environment to achieve deterministic traffic shaping with $O(1)$ complexity.

Based on the trade-off between steady-state error and response speed in control theory, we suggest using the following empirical formula to set backpressure thresholds:

$$T_{warn} = 2 \cdot \overline{N}_{processed},\ T_{critical} = 5 \cdot \overline{N}_{processed} \tag{5}$$

Where $\overline{N}_{processed}$ is the average number of tasks the system can process per unit time in a steady state. This setting avoids frequent triggering of backpressure while ensuring the system does not lose stability under burst loads.

5. Experiments and Evaluation

5.1. Experimental Setup

Hardware Environment: To verify the architecture's universality, experiments were conducted in a virtualized environment simulating an edge computing node (WSL2, limited to 2 vCPU / 2GB RAM).

Scenario: The task was a day/night dual-mode security inference (ResNet-18 + BiLSTM), which involves multimodal dynamic routing as exemplified in Figure 4.

Comparison Groups: The Baseline Group used standard TCP Loopback communication with simulated Ext4 disk storage, while the Experimental Group utilized the SlotFlow architecture based on tmpfs in-memory file slots.

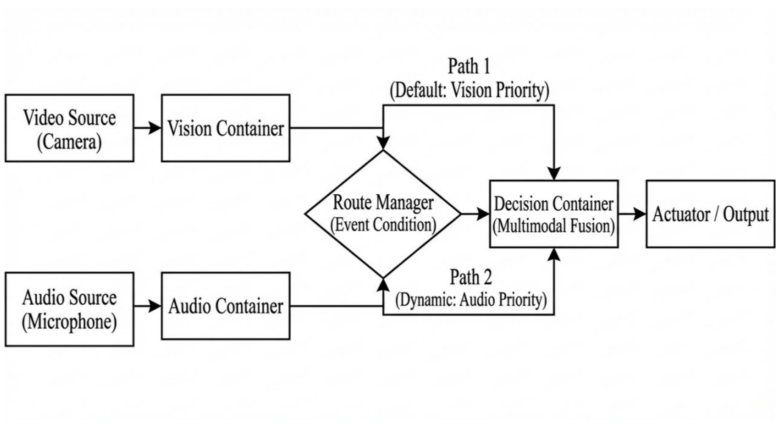
Path 1
(Default: Vision Priority)
Video Source (Camera)
Vision Container
Route Manager (Event Condition)
Decision Container (Multimodal Fusion)
Actuator / Output
Path 2
(Dynamic: Audio Priority)
Audio Source (Microphone)
Audio Container

Figure 4: Multimodal Dynamic Routing Scenario Example

5.2. I/O Latency and Throughput Analysis

As benchmarked in Figure 5, in terms of communication latency, the native Linux kernel shows that SlotFlow's Ping-Pong round-trip latency is as low as 24.6 µs. This represents a reduction of approximately 84% compared to the 150.3 µs for TCP Loopback. Although file system overhead in the WSL2 virtualization environment causes fluctuations in absolute values, the relative advantage brought by its zero-copy mechanism remains significant.

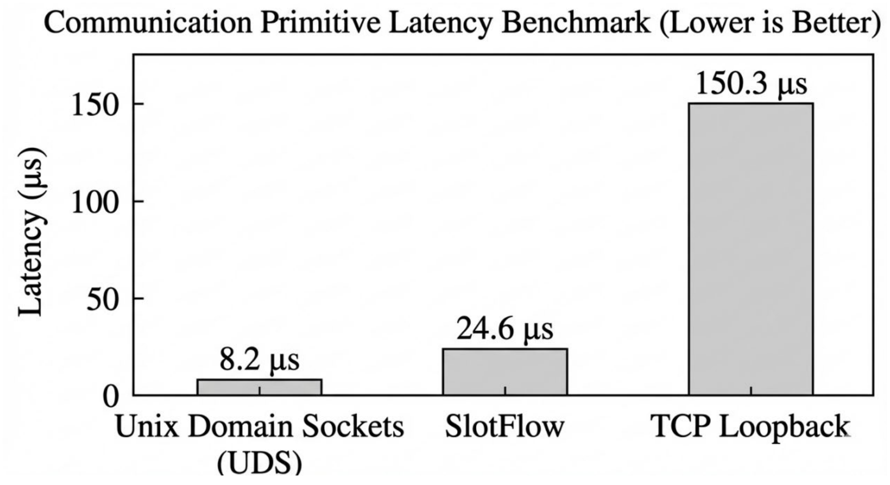

Figure 5: Communication Primitive Latency Benchmark

In throughput and I/O wait tests, the advantage is particularly evident. We simulated continuous high-frequency writing of 1000 frames (simulating a 100 FPS video stream). As shown in Table 1, the experimental data indicate:

Table 1: Performance Comparison

Metric	Ext4 (Simulated SD Card)	SlotFlow (Memory Slot)	Improvement
Total Time	18.75 s	11.45 s	+38.9%
Bottleneck Source	Disk I/O Blocking	Pure CPU Computing	—

Ext4 Disk Group: Due to frequent fsync flush operations, processing 1000 frames took 18.75 seconds, with significant I/O wait (iowait), preventing the CPU from working at full load.

SlotFlow Group: Benefiting from memory operations, the same task took only 11.45 seconds.

Conclusion: SlotFlow improved processing efficiency by approximately 38.9%, effectively eliminating the I/O bottleneck in edge inference.

5.3. Backpressure Protection and Robustness Verification

To verify system stability, we constructed a "Producer-Consumer" rate mismatch scenario: The producer generates tasks at a rate of 30 FPS, while the consumer can only process at 5 FPS due to simulated load lag. The experimental results are shown in Figure 6:

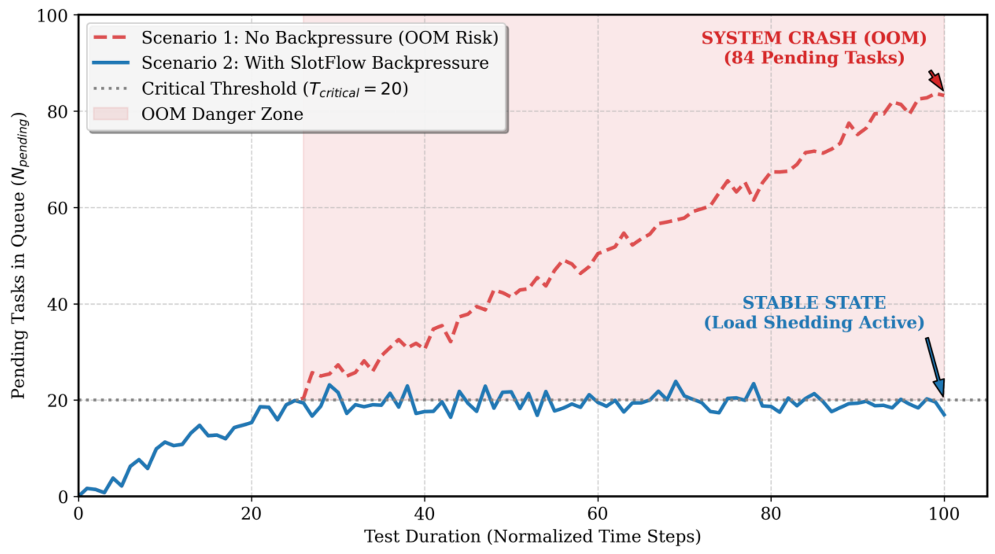

Figure 6: Comparison of Task Queue Backlog under Overload Conditions

Scenario 1 (No Backpressure): The queue builds up rapidly, eventually leading to a service crash (CRASH) or OOM.
Scenario 2 (SlotFlow Backpressure): The system maintains a stable state by triggering load shedding/early exit when the critical threshold is reached.

6. Discussion

6.1. Cross-Node Scalability

SlotFlow currently targets collaboration within a single node. For cross-node communication (e.g., Cloud-Edge collaboration), which typically involves more complex network conditions and model splitting strategies as described by Li et al. [2] and Matsubara et al. [8], we suggest a hybrid architecture: Use SlotFlow within the node to achieve extreme performance, and bridge to ZeroMQ or MQTT via an edge gateway for inter-node communication, combined with fine-grained model partitioning algorithms for task scheduling.

6.2. Comparison with Stream Processing Systems

Compared to heavy stream processing systems like Apache Flink or Kafka, SlotFlow is much more lightweight. Sedlak et al. [9] studied the autoscaling of stream processing on edge devices and found that the startup and maintenance overhead of heavy frameworks is huge. SlotFlow sacrifices some advanced features (such as Exactly-once semantics) in exchange for extremely low latency and resource consumption on low-power devices, making it more suitable for real-time multimodal AI tasks.

6.3. Security

Shared memory does reduce isolation. However, in SlotFlow, container privileges can be restricted via User Namespaces and Seccomp, opening only necessary file operations to strike a balance between performance and security.

7. Conclusions

The SlotFlow architecture proposed in this paper effectively solves the I/O bottlenecks, storage wear, and overload crash problems in traditional edge orchestration by combining in-memory file systems, metadata-driven routing, and dynamic backpressure protection. By introducing atomic counters, hybrid polling mechanisms, and a strict Swap prohibition strategy, SlotFlow guarantees high performance while significantly enhancing system robustness. Experiments have proven its efficiency in single-node multimodal collaboration scenarios. Future work will focus on researching the distributed consistency extension of SlotFlow in cross-node clusters and support for zero-copy sharing of GPU memory.

References

[1] Pan, A., & Ray, P. P. (2025). LLMYOLOEdge: An edge-IoT aware novel framework for integration of YOLO with localized quantized large language models. IEEE Access, 13, 167250–167279. https://doi.org/10.1109/ACCESS.2025.11176056

[2] Li, H., Li, X., Fan, Q., He, Q., Wang, X., & Leung, V. C. M. (2024). Distributed DNN inference with fine-grained model partitioning in mobile edge computing networks. IEEE Transactions on Mobile Computing, 23(10), 9060–9074. https://doi.org/10.1109/TMC.2024.3357346

[3] Yakubov, D., & Hästbacka, D. (2025). Comparative analysis of lightweight Kubernetes distributions for edge computing: Performance and resource efficiency. In Service-Oriented and Cloud Computing. ESOCC 2025 (pp. 1–22). Springer. https://www.researchgate.net/publication/390570884_Comparative_Analysis_of_Lightweight_Kubernetes_Distributions_for_Edge_Computing_Performance_and_Resource_Efficiency

[4] Alqaisi, O. I., Tosun, A. Ş., & Korkmaz, T. (2024). Performance analysis of container technologies for computer vision applications on edge devices. IEEE Access, 12, 41852–41869. https://doi.org/10.1109/ACCESS.2024.3375836

[5] Gupta, R., & Nahrstedt, K. (2025). Performance characterization of containers in edge computing (arXiv:2505.02082). arXiv. https://arxiv.org/abs/2505.02082

[6] Oh, S., Kim, J., Han, S., Kim, J., Lee, S., & Noh, S. H. (2024). MiDAS: Minimizing write amplification in log-structured systems through adaptive group number and size configuration. In Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST 24) (pp. 1–16). USENIX Association. https://www.usenix.org/conference/fast24/presentation/oh

[7] Ekane, B., Mvondo, D., Lachaize, R., & Bromberg, Y.-D. (2025). DiSC: Backpressure mitigation in multi-tier applications with distributed shared connection. In Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25) (pp. 55–70). USENIX Association. https://www.usenix.org/conference/nsdi25/presentation/ekane

[8] Matsubara, Y., Levorato, M., & Restuccia, F. (2023). Split computing and early exiting for deep learning applications: Survey and research challenges. ACM Computing Surveys, 55(5), Article 90. https://doi.org/10.1145/3527155

[9] Sedlak, B., Raith, P., Morichetta, A., Casamayor Pujol, V., & Dustdar, S. (2025). Multi-dimensional autoscaling of stream processing services on edge devices (arXiv:2510.06882). arXiv. https://arxiv.org/abs/2510.06882