OQueue: Observable Communication in Learning Directed Operating Systems

Aditya Atul Tewari
adityaatewari@utexas.edu
University of Texas at Austin

Sujay Yadalam
sujayyadalam@cs.wisc.edu
University of Wisconsin-Madison

Arthur Michener Peters
amp@cs.utexas.edu
University of Texas at Austin

Saurabh Agarwal
agarwal@cs.wisc.edu
University of Texas at Austin

Aditya Akella
akella@cs.utexas.edu
University of Texas at Austin

Michael M. Swift
swift@cs.wisc.edu
University of Wisconsin-Madison

Christopher J. Rossbach
rossbach@cs.utexas.edu
University of Texas at Austin,
Microsoft Research

Abstract

The need for rapid policy development and Machine Learning (ML) to catch up with the increasing demands on Operating Systems made by evolving hardware and applications has exposed the lack of a data-plane abstraction for efficient data movement between OS subsystems. OQueues are a new data-plane abstraction designed to aid in flexible data-driven policy design. They express a novel *observer paradigm*, utilizing weak observers for history introspection and strong observers for exactly-once view semantics. We evaluate the utility and scalability of OQueues and use them to implement several ML-based policies for a file-prefetcher.

ACM Reference Format:
Aditya Atul Tewari, Sujay Yadalam, Arthur Michener Peters, Saurabh Agarwal, Aditya Akella, Michael M. Swift, and Christopher J. Rossbach. 2025. OQueue: Observable Communication in Learning Directed Operating Systems. In *Practical Adoption Challenges of ML for Systems (PACMI '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, Koblenz, Germany, 6 pages. https://doi.org/10.1145/3766882.3767174

1 Introduction

Machine Learning (ML) driven policies have shown significant benefits in fields such as cloud computing [3], databases [21], networking [1, 22], and CDN management [5]. However, Operating Systems (OSes) rely on myriad resource-management policies which have yet to take advantage of ML, despite showing significant promise [1, 2, 4, 6, 8, 9, 11]. We believe that one primary reason data-driven ML policies have not been widely adopted for OSes is the friction between the data requirements of ML policies and the current structure of OSes. This makes providing data to such policies a challenge.

Consider a file system prefetcher that predicts when and where an access to a file location will occur. Predicting accesses can depend on many sources of information, such as the state of the scheduler, virtual memory, caches, or disk. Currently, ensuring information from different OS subsystems is easily available, or *observable*, to a data-driven policies requires modifications to the OS kernel, which can be cumbersome and error-prone. This becomes untenable as multiple data-driven policies are added.

We believe that to increase adoption of ML-driven OS policies, a unified data-plane is required to facilitate cross-subsystem state observation. Such a data-plane should deliver the following properties: (i) enable observation of data across subsystem boundaries with minimal code restructuring; (ii) support notifications, enabling developers to subscribe to data sources and receive notifications when data is available; (iii) support historical querying of data, for example, getting the last five file accesses when building a file-prefetcher; (iv) restrict the mode of all communication to message passing semantics, while ensuring implementations are as efficient as shared memory.

We propose a new primitive that provides these capabilities — Observable Queues (OQueues). OQueues unify communication and observation into a single primitive, providing kernel policy developers with observability across different subsystems. Policies can then access whichever OQueue has the data they desire, instead of manually modifying each subsystem to capture desired data.

To achieve these properties, OQueues introduce the new concept of *observers* to the standard producer-consumer paradigm. Observers, instead of consuming items from a queue, can inspect the history of messages or receive messages as they are produced. Support for observers enables different

PACMI '25, October 13–16, 2025, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2205-9/25/10
https://doi.org/10.1145/3766882.3767174

subsystems to have access to each other's data when making decisions.

In this paper, we discuss the design, implementation, and use of this primitive in the OS. A preliminary, unoptimized implementation of OQueues shows an average-case overhead of 12% over that of multi-producer, multi-consumer queues, due to supporting observers. We find that overheads remain largely constant as additional policies or data collection are introduced. Finally, we show a proof-of-concept by implementing a file-prefetcher with two learned policies.

2 Background & Motivation

Learned policies for OS. Prior work [1, 7, 9, 11] has studied the use of ML to make OS policy decisions. Orca [1] makes decisions about the TCP congestion window and delay to improve network utilization. Heimdall [11] and LinnOS [9] make decisions about routing I/O requests to disk. MLLB [4] examines load balancing for scheduling in the Linux kernel. Klieo [6] examines memory tiering for heterogeneous memory. Overall, these studies demonstrate the potential benefit for ML-driven policies in the OS.

However, each of these studies focuses on a single subsystem and. None suggest or provide a data-plane abstraction for cross-subsystem data movement required for learned policies, since supporting such behavior was not their goal.

Support for building Learned policies. LAKE [8] and KMLib [2] are frameworks for deploying ML-based policies in the OS. Both systems fall short of enabling cross-subsystem data views for ML deployment.
KMLib: KMLib [2] provides mechanisms to run models in the kernel but forces the policy developer to implement the data movement, significantly reducing developer productivity.
LAKE: LAKE [8] introduces a feature store to enable cross-subsystem observation. LAKE's feature store does not meet developer needs for multiple reasons. The API is focused on data capture and query semantics, and support for notifications was a non-goal. Any attempt to add notifications requires using *notification chains*, a Linux feature to call callbacks registered by other subsystems. Using notification chains will lead to data duplication because data must be copied to the chain and LAKE feature store, separately. Similarly, supporting consumers will require explicit queues to allow mutually exclusive access to data, leading to data duplication. Further, LAKE requires kernel recompilation each time a new data capture point is created, making the introduction of dynamic policies challenging. Prime facie, rearchitecting LAKE's feature store using ring-buffers on top of eBPF arrays could resolve some of the kernel recompilation challenges. However, it would not fix the lack of support for notifications and consumers, which require the code changes described above. These code changes require kernel recompilation and reboot, slowing development.

Need for a cross-subsystem data plane abstraction A cross-subsystem data plane abstraction is required to enable models that depend on the state of multiple subsystems. For example, MLLB benefits from information about how processes interact with NUMA nodes and information about timeslices on the specific processes. [4] Without explicit support for cross-subsystem data sourcing, this information is sequestered behind module boundaries, and making it accessible requires code modification. This means that the kernel needs to be recompiled and the OS rebooted, resulting in significantly slower development of new policies and application downtime.

Design Goals for data-plane abstractions. To support ML in the kernel, we outline design goals for data-plane abstractions that enable ML support and are useful for subsystem communication. These goals are: (i) support for notifications; (ii) support for historical data queries; (iii) ubiquitous message passing. Notification support allows dynamic insertion of policy execution because APIs that access and change data are interposable. Historical data queries are required for several ML models such as Heimdall and LinnOS [8, 9, 11, 12], which use information about the last several I/O requests. Finally, message passing must be ubiquitous in a data-plane abstraction; otherwise, subsystems may exchange data outside the view of tracking mechanisms.

OQueues provide kernel developers a simple data-plane abstraction that meets these design goals. OQueues further ensure that overheads to a particular OQueue with no consumers and no other subsystems registered for notifications is negligible. Emphasis on parsimony makes OQueues efficient while achieving each the design goals.

3 OQueues

We first provide an example subsystem we would like to improve with ML and OQueues, then describe OQueues's design, its declarative interface, and implementation.

3.1 Preliminary Example and Terms

In a file system (FS), programs make requests to an API, which in turn makes requests to a page cache, which may make requests to disk. A prefetcher can improve its performance by prefetching data into the page cache. To inform its decisions, the prefetcher should be able to receive a notification each time the FS requests something from the page cache and view a history of previous requests made by the FS. In figure 1, we illustrate the data flow between the prefetcher, FS, scheduler, and page cache. We describe the components of that dataflow when using OQueues below.

Policies Policies process data from consumers and observers and produce values that are either used to make decisions directly or fed to other policies in the system.

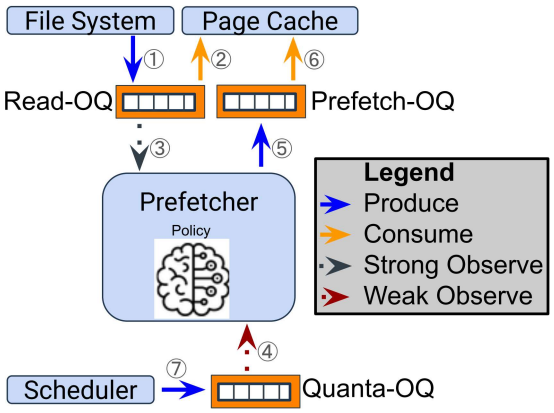


Figure 1. A file system (FS) prefetcher using OQueues. ①: An FS produces a read request via the Read-OQ. ②: A page cache consumes a request from the Read-OQ and performs the read. ③: The prefetcher strongly observes this Read-OQ request and determines what the next prefetch should be, if any. ④: To source data for the prefetcher's policy, it asks for the history of the last few processes that ran and their quanta end time from the Quanta-OQ. ⑤: The prefetcher produces a request to the Prefetch-OQ. ⑥: The page cache reads that prefetch request and executes it. ⑦: The quanta runtime was produced for use by the scheduler, and the history was persisted only when ④ was declared by the prefetcher.

Consumers and Producers Producers enqueueing values, while consumers dequeueing values. Together these provide the simple abstraction of a messaging queue. Arrows ① and ② in figure 1 show an example: the file system produces read requests consumed by the page cache.

Strong Observers Strong observers provide a mechanism to allow exactly-once message views. The primary use of strong observers is to ensure that certain policies run when a particular piece of data changes. A prefetcher policy that should be triggered when a file read request occurs, as in figure 1 ③, can register itself to strongly observe the read request and thus run after the new request occurs. Additionally, multiple strong observers can observe the same value exactly once each. If a particular consumer dequeues a value before a strong observer gets its chance, the value is persisted until that observer consumes it.

Weak Observers In contrast to strong observers, weak observers do not support exactly-once message views. Instead, they allow for queries of the history of a particular OQueue, including values that may have already been consumed or observed. In the prefetcher example, if the policy wanted to employ a history-based ML model, it could use the last 10 quanta runtime to determine the next prefetch. Weak observation provides this history as shown in figure 1 ④.

3.2 Declarative Interfaces

OQueues require that each producer, consumer, and observer explicitly attach themselves before using the OQueue. This allows each OQueue to dynamically respond to each attachment by optimizing the OQueue. Thus, if no observers or consumers are present, no data is ever captured. While OQueues might be ubiquitous in the system, costs for them are only incurred when the data is required by a consumer or observer. In our file-prefetcher example, if a new prefetcher policy desired a history of accesses, OQueue must declare the size of the history it was interested in when declaring a weak observer, see table 1 for details. This enables efficient capture and storage of only features currently in use.

3.3 Implementation

We implement OQueues on top of lock-free ring-buffers in Linux in userspace and in Asterinas [15] in kernel space, which allows consumer and producer semantics. Then, to support observers, our Rust implementation forces messages to implement the `Copy`, `Clone`, `Sync`, and `Send` traits. To support strong observers, there is an additional pointer within the queue for each observer. This pointer is used to ensure that the observer reads all messages exactly once before they are deleted. To implement weak observers, we employ an atomically packed valid flag and version that are used by reading the flags, reading data, and finally validating the flags to ensure the data is valid.

3.4 Discussion

OQueue as a synchronization primitive. OQueue implementations must wait for data or space to be available in various situations. Currently we busy-wait, but there are various ways to synergize with OS schedulers, similar to the Synthesis Kernel [13]. For example, the scheduler could be OQueue-aware, and the OQueue itself could act as a synchronization primitive between kernel threads. The kernel could have a specially optimized path for transferring control directly from a producing to a consuming thread. It would also be possible to have enqueue operations directly invoke waiting consumers. OQueue-aware scheduling would reduce the overhead of message-based communication.

Temporal Queries on OQueues. While OQueues provide a powerful primitive to make model deployment and data-pipeline iteration easier and faster, observer semantics can admit a more powerful interface. A Temporal Query Language (TQL) [10] is generally used to search through time-series data. We propose allowing observers to predefine data queries using a TQL. The goal is for OQueue to reduce data capture overheads by introducing checks at the producer site for the disjunction of all subscribed observers' queries. By not enqueuing those data elements, the producer would not incur the overheads of enqueuing unused data. Additionally, extending OQueues to a TQL query interface would reduce programmer effort when transforming raw

Table 1. The OQueue API. `put`, `take`, and `strong_observe` are blocking. Non-blocking variants are available but omitted for brevity.

API	Description
`OQueue::attach_producer() -> Producer`	Get a producer handle
`OQueue::attach_consumer() -> Consumer`	Get a consumer handle
`OQueue::attach_strong_observer() -> StrongObserver`	Get a strong observer handle
`OQueue::attach_weak_observer(n: int) -> WeakObserver`	Get a weak observer handle specifying the max size of history the observer is interested in
`Producer::put(value: T)`	Enqueue a message
`Consumer::take() -> T`	Dequeue a message
`StrongObserver::strong_observe() -> T`	Observe a message from the queue
`WeakObserver::history(n: int) -> [T]`	Get the most recent `n` messages in the queue

data into ML tensors, because query languages are the underlying abstraction of dataframes, a familiar interface for most ML engineers.

Simplifying MLOps with OQueues. MLOps [14] is the cycle of development used for fast ML iteration. It includes a *Data* collection phase, *Model* development phase, and a *Deploy* phase. OQueues with TQL abstract the direct manipulation of the system by the developer because the desired data can be materialized when needed during feature selection and the *Model* phase. Consider that we could view the data in each OQueue as a column in a massive virtual database; then TQL queries over this database would only materialize the exact data required to train a model or compare features. This TQL query tracking reduces developer effort during the *Data* phase of the MLOps cycle.

4 Evaluation

To evaluate OQueues, we measure their scalability as communication primitives and iteratively improve a prefetcher with them.

Setup. All testing was performed on Linux 6.12.10 (benchmarking an OQueue-aware kernel is future work). All runs were on AMD Ryzen 9 8945HS (8c/16t) with SMT on.

We measure OQueue performance using an RPC or "ping-pong" microbenchmark between threads. This benchmark represents the common case among communicating subsystems; therefore, it is especially important for performance. In particular, performance is sensitive to how control is transferred from producers to consumers and observers.

We measure two mechanisms: Busy-waiting by executing an empty loop, and Yield-waiting by looping on an OS scheduler yield. We measured OQueue performance and scalability compared against state-of-the-art "rigtorp" message queues [16, 17], which only support busy-waiting. To measure the overhead of using the OQueue data structure without any control transfer overhead, we measured the cost of passing an argument to a function, 7 ns, vs. the same done after placing the argument in an OQueue, 11 ns. Thus, the data structure overhead is small for inter-subsystem calls and demonstrates an ideal transfer of control mechanism.

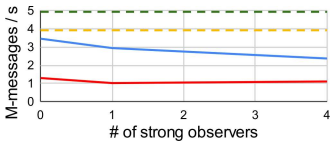
RPC for Strong Observers As shown in figure 2a, increasing the number of strong observers does not significantly impact message throughput. Busy-waiting has inferior scaling as it uses a full core for each thread, forcing many accesses to transfer data between cores. This increases contention even when only one core can progress. Yield-waiting has higher latency but better scaling, due to the cost of context-switches and avoiding contention between threads. We believe an OQueue-aware scheduler could keep latencies close to busy-waiting with improved scaling.

We also compare against the "rigtorp" MPMC [16] and SPSC [17], showing a 12% slow-down between our implementation and state-of-the-art queues which do not support observers. We have not implemented all of the optimizations present in "rigtorp", so we believe reaching the performance of at least MPMC "rigtorp" is possible. All measurements use ring-buffers of length 2 because RPC does not have more than one message in flight at a time.

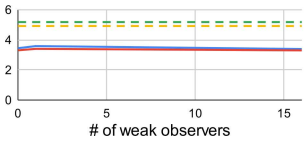
RPC for Weak Observers As shown in figure 2b, RPC throughput remains constant with an increase in weak observers. The data is shown for weak observers accessing histories of length 2 and 64 (using corresponding ring-buffer sizes). Each weak observer runs every 10 ms and reads the entire available history (2 or 64), meaning that weak observers perform more work with a history of 64, likely accounting for the small decrease in performance.

4.1 Using OQueue for FS Prefetching

We perform a case study on file-prefetching using OQueues on Asterinas [15], a monolithic kernel implemented in Rust. Asterinas includes a cache per file, and we add OQueues to Asterinas to collect data. We create two ML-driven policies that are deployed as prefetchers. The model within each ML-driven policy predicts the next file offset to prefetch. We train a perceptron which attains 99% accuracy with 2 weights and 1 bias, and we train a linear regression model. We use a microbenchmark deploying either a data collector "policy" for data collection for training or an actual prefetching policy during model deployment, thereby specializing these models to this set of accesses. We also deploy a next-page prefetcher to express how simple it is to make these changes.



(a) RPC throughput with increasing strong observers. All measurements use ring buffers of length 2.



(b) RPC throughput with increasing weak observers at multiple history lengths. Each weak observer runs every 10 ms. Blue and yellow are length 2, others are length 64.

Figure 2. Solid red and blue are OQueue busy-wait and yield, resp. Dashed yellow and green are "rigtorp" MPMC and SPSC, resp.

Data Collection To collect data, we create a strong observer on the OQueue that records the file reads we are interested in observing. This same data is used to train all the remaining policies. We also create a strong observer that records scheduler quanta to a file. However, this information had no effect on model performance and thus was not used.

Deploying Policies using OQueues The file-prefetcher policy listens on an OQueue for file operations as a strong observer and makes a decision about what to prefetch, enqueuing that decision to the OQueue the file system uses to perform prefetches. Once OQueues have been added to Asterinas' prefetcher, the process of deploying a new policy only requires 10s of LoC and less than an hour per policy.

4.2 Results

We deploy a heuristic policy and two learned policies in Asterinas. We also compared against no prefetching and a sequential prefetcher already available in Asterinas. Table 2 shows the performance increase and improved cache hit rate created by all policies. These measurements demonstrate the range of policies that can be implemented rapidly. The sequential prefetcher runs inside the page cache code and stores its state in the page cache. The OQueue strided heuristic prefetcher uses a strong observer to receive notifications for reads, and then a weak observer on the read history to detect strided accesses by an individual thread. The OQueue regression and perceptron prefetchers only use the most recent read.

Comparison to other Frameworks. We compare the lines of code required for our microbenchmark to take the same features compared to various systems such as LAKE, KMLib, and Heimdall, refer to table 3 for details [2, 8, 11]. LAKE does not support feature capture during inference within its artifact. Instead its benchmarks do not use its feature store. Heimdall takes over 50 LoC for its bespoke data movement out of the block_io subsystem. With OQueue however, it takes 7 LoC to persist data to the file system and 4 LoC for inference, due to OQueue's declarative interface.

Table 2. Comparison of prefetcher performance using OQueue (OQ) and others.

Prefetcher	Time	Hit rate
None	17.3 s	0.0%
Sequential	17.3 s	0.1%
OQ heuristic	16.4 s	49.0%
OQ regression	16.0 s	57.5%
OQ perceptron	16.0 s	57.3%

Table 3. Lines of Code comparison for various systems, moving the same data.

System	Collection	Inference
LAKE	62 LoC	No support
KML	Traces	No Support
Heimdall	62 LoC	53 LoC
OQueue	7 LoC	4 LoC

5 Related Work

Other systems exist for data-plane management, both in and out of the kernel. However, to our knowledge, no other system provides the semantics and performance potential of OQueues.

Kafka Kafka is a platform for event streaming [18]. It lets producers publish events, and consumers subscribe to "topics" to observe data. Kafka consumers have the semantics of OQueue's strong observers because they can be grouped such that messages are delivered to only one consumer, but messages are assigned to a consumer when sent (published into a specific partition). This could result in a consumer waiting for a message while one is available in a different partition, violating the semantics of a queue.

ROS Robot Operating System (ROS) is an extension of Linux for robot controllers [20]. ROS includes a communication primitive called "topics" which implement strong observer semantics but neither consumers nor weak observers.

DBOS Database Operating System (DBOS) uses a database for all communication [19]. Database tables are similar to OQueues but cannot dynamically elide data collection or storage because database interfaces do not require policies to declare their interest ahead of time. Further, DBOS implements message consumption by deleting the message from the table thus, it will no longer be available for observation.

6 Conclusion

In this work, we introduced OQueues, a new data-plane abstraction for communication between OS subsystems. OQueues are ubiquitously deployed throughout the OS and respond dynamically to new observers and consumers. To achieve this, OQueues ensure that only observed/consumed data are ever created in the first place. With the aid of OQueues, policy developers can easily deploy and iterate over ML-driven OS policies, while metadata manipulation is automatically optimized for only active policies.

7 Acknowledgments

This material is based on work supported by the U.S. National Science Foundation (NSF) under Grant Number 2326576. A special thanks to Emily May Peterson for her help in debugging the first implementation of weak observers.

References

[1] Abbasloo, S., Yen, C.-Y., and Chao, H. J. Classic meets modern: a pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 632–647.

[2] Akgun, I. U., Aydin, A. S., and Zadok, E. KMLIB: Towards machine learning for operating systems.

[3] Alsubaei, F. S., Hamed, A. Y., Hassan, M. R., Mohery, M., and Elnahary, M. K. Machine learning approach to optimal task scheduling in cloud communication. *Alexandria Engineering Journal 89* (2024), 1–30.

[4] Chen, J., Banerjee, S. S., Kalbarczyk, Z. T., and Iyer, R. K. Machine learning for load balancing in the linux kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2020), APSys '20, Association for Computing Machinery, pp. 67–74.

[5] Chen, J., Sharma, N., Khan, T., Liu, S., Chang, B., Akella, A., Shakkottai, S., and Sitaraman, R. K. Darwin: Flexible learning-based cdn caching. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA, 2023), ACM SIGCOMM '23, Association for Computing Machinery, p. 981–999.

[6] Doudali, T. D., Blagodurov, S., Vishnu, A., Gurumurthi, S., and Gavrilovska, A. Kleio: A hybrid memory page scheduler with machine intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2019), HPDC '19, Association for Computing Machinery, p. 37–48.

[7] Doudali, T. D., Blagodurov, S., Vishnu, A., Gurumurthi, S., and Gavrilovska, A. Kleio: A hybrid memory page scheduler with machine intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2019), HPDC '19, Association for Computing Machinery, pp. 37–48.

[8] Fingler, H., Tarte, I., Yu, H., Szekely, A., Hu, B., Akella, A., and Rossbach, C. J. Towards a machine learning-assisted kernel with lake. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 846–861.

[9] Hao, M., Toksoz, L., Li, N., Halim, E. E., Hoffmann, H., and Gunawi, H. S. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 173–190.

[10] Jonker, A., and Mucci, T. What is structured query language (sql)?, Jul 2025.

[11] Kurniawan, D. H., Putri, R. A., Qin, P., Zulkifli, K. S., Sinurat, R. A. O., Bhimani, J., Madireddy, S., Kistijantoro, A. I., and Gunawi, H. S. Heimdall: Optimizing storage i/o admission with extensive machine learning pipeline. In *Proceedings of the Twentieth European Conference on Computer Systems* (New York, NY, USA, 2025), EuroSys '25, Association for Computing Machinery, p. 1109–1125.

[12] Li, X., Wu, G., Yang, L., Wang, W., Song, R., and Yang, J. A survey of historical learning: Learning models with learning history, 2023.

[13] Massalin, H., and Pu, C. Threads and input/output in the synthesis kernal. *SIGOPS Oper. Syst. Rev. 23*, 5 (Nov. 1989), 191–201.

[14] Matsui, B. M. A., and Goya, D. H. Mlops: five steps to guide its effective implementation. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI* (New York, NY, USA, 2022), CAIN '22, Association for Computing Machinery, p. 33–34.

[15] Peng, Y., Tian, H., Zhang, J., Li, R., Chen, C., Jiang, J., Xian, J., Wang, X., Xu, C., Zhou, D., Luo, Y., Yan, S., Zhang, Y., and SUSTech. Asterinas: A linux abi-compatible, rust-based framekernel os with a small and sound tcb. *Proceedings of the USENIX Annual Technical Conference* (2025).

[16] Rigtorp, E. MPMCQueue. https://github.com/rigtorp/MPMCQueue.

[17] Rigtorp, E. SPSCQueue. https://github.com/rigtorp/SPSCQueue.

[18] Sax, M. J. *Apache Kafka.* Springer International Publishing, Cham, 2020, pp. 1–8.

[19] Skiadopoulos, A., Li, Q., Kraft, P., Kaffes, K., Hong, D., Mathew, S., Bestor, D., Cafarella, M., Gadepally, V., Graefe, G., Kepner, J., Kozyrakis, C., Kraska, T., Stonebraker, M., Suresh, L., and Zaharia, M. Dbos: A dbms-oriented operating system. *Proceedings of the VLDB Endowment (PVLDB) 15*, 1 (2022), 21–30.

[20] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

[21] Van Aken, D., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Bilien, C., and Pavlo, A. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proc. VLDB Endow. 14*, 7 (Mar. 2021), 1241–1253.

[22] Wu, D., Wang, X., Qiao, Y., Wang, Z., Jiang, J., Cui, S., and Wang, F. Netllm: Adapting large language models for networking. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 661–678.