Building ultra-performant message streaming

SUB-MS P99+ LATENCY · MULTI GB/S THROUGHPUT

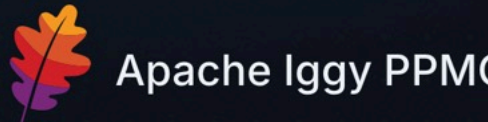
sub-ms p99+ | **multi GB/s** throughput | **Apache Iggy** inside

Piotr Gankiewicz | laserdata | Apache Iggy PPMC

laserdata | IGGY

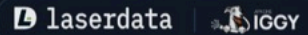
Hello!

laserdata



spetz

spetzu



Why Message Streaming?

When you need an immutable log of strictly ordered records

When persistence and replayability truly matter

When high throughput and low latency are required

When multiple consumers independently read the same data

...you might need a log.

The Append-Only Log

"

The truth is the **log**.

The database is a **cache**

of a **subset** of the **log**.

laserdata | IGGY

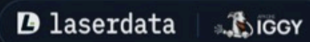
#0	#1	#2	#3	#4	#5	#6

offset
3

payload
{ "temp": 23.5 }

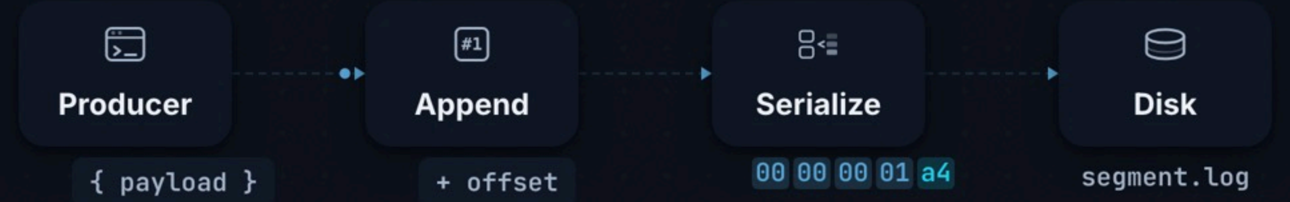
metadata
timestamp | id | length

```rust
struct Stream {
    name: String,
    offset: u64,
    messages: Vec<Message>,
}

fn append(&mut self, payload: Vec<u8>) {
    let msg = Message::new(self.offset, payload);
    self.messages.push(msg);
    self.offset += 1;
}

fn poll(&self, offset: u64, count: u64) -> &[Message]
    &self.messages[offset..offset + count]
}
```



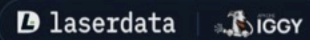
Writing Data

Serialize to raw bytes and flush to disk



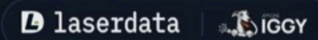
ON-DISK BINARY LAYOUT

offset	timestamp	id	length	payload
8B	8B	16B	4B	N B
0	8	16	32	36..N



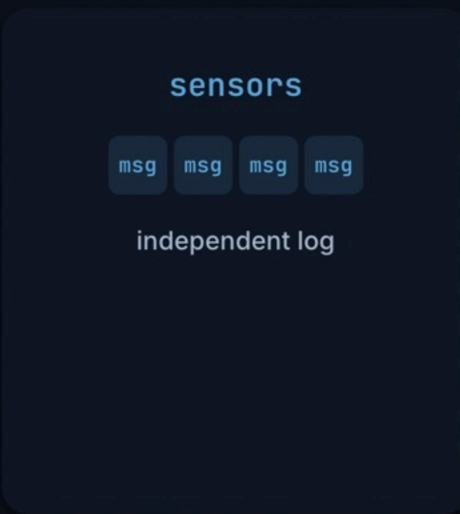
Reading Data

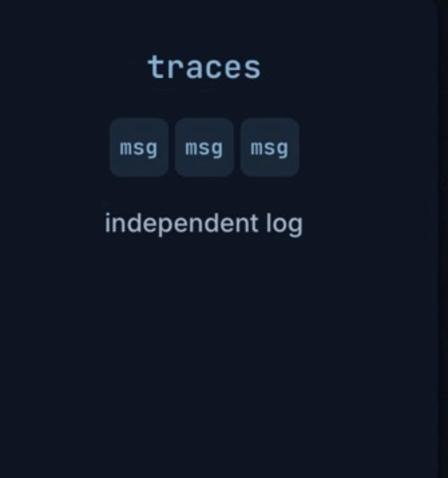
Consumer polls, server reads from disk



One Log to Rule Them All?

Separate them by category - Topics

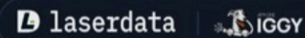




Split Into Partitions

Parallel append-only logs within one topic

- Independent append-only logs
- Scale writes horizontally
- More partitions = more throughput



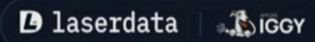
Where Does the Data Live?

Index files map offsets to byte positions - instant lookup

SEG_002.INDEX

OFFSET	BYTE POSITION	
0	0x0000	jump here
1	0x00A4	
2	0x0148	
3	0x01EC	
4	0x0290	

Index lookup, no full scan

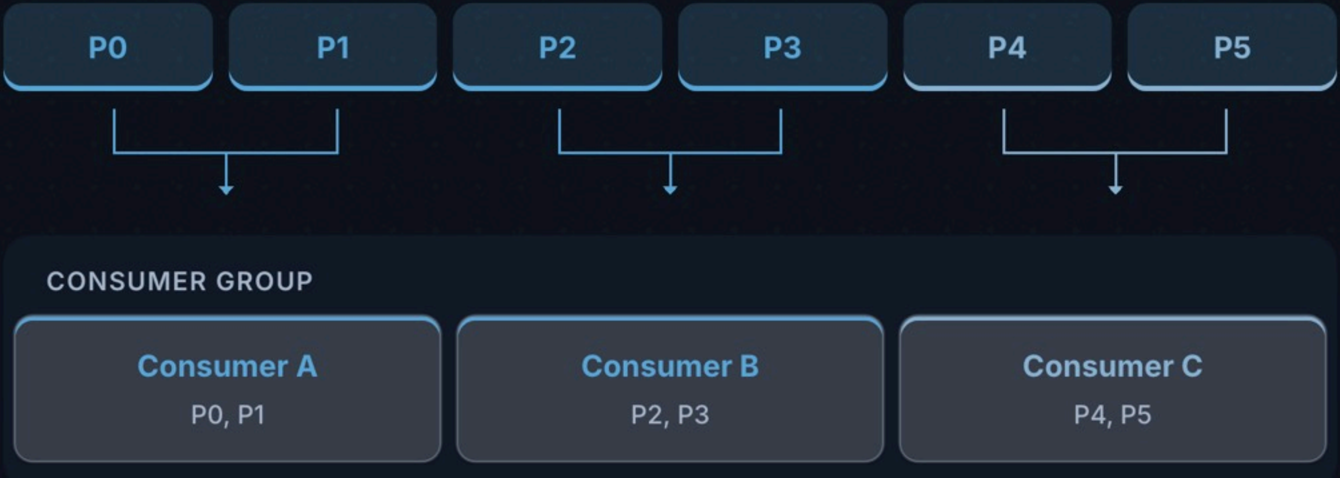


Putting It All Together

Stream / Topic / Partition / Segment

Consumer Groups

Each partition assigned to exactly one consumer in a group



Each partition processed by exactly one consumer

Are we done yet?

Not even close.

How do we serialize without copying?

How do we flush data safely to disk?

How do we do async I/O properly?

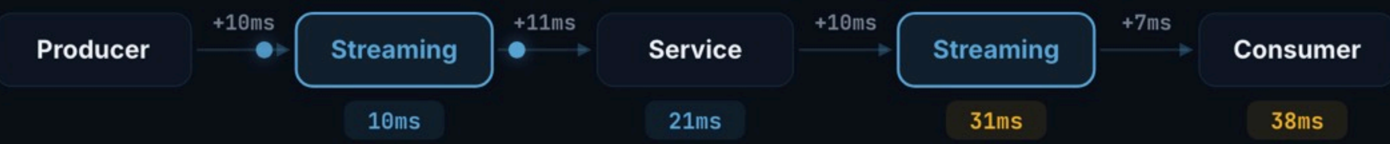
How do we use all CPU cores?

Why even bother?

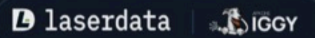
Every Millisecond Counts.

In distributed systems, **latency compounds**.
Every service call, every hop, every layer **adds up**.

340 ms

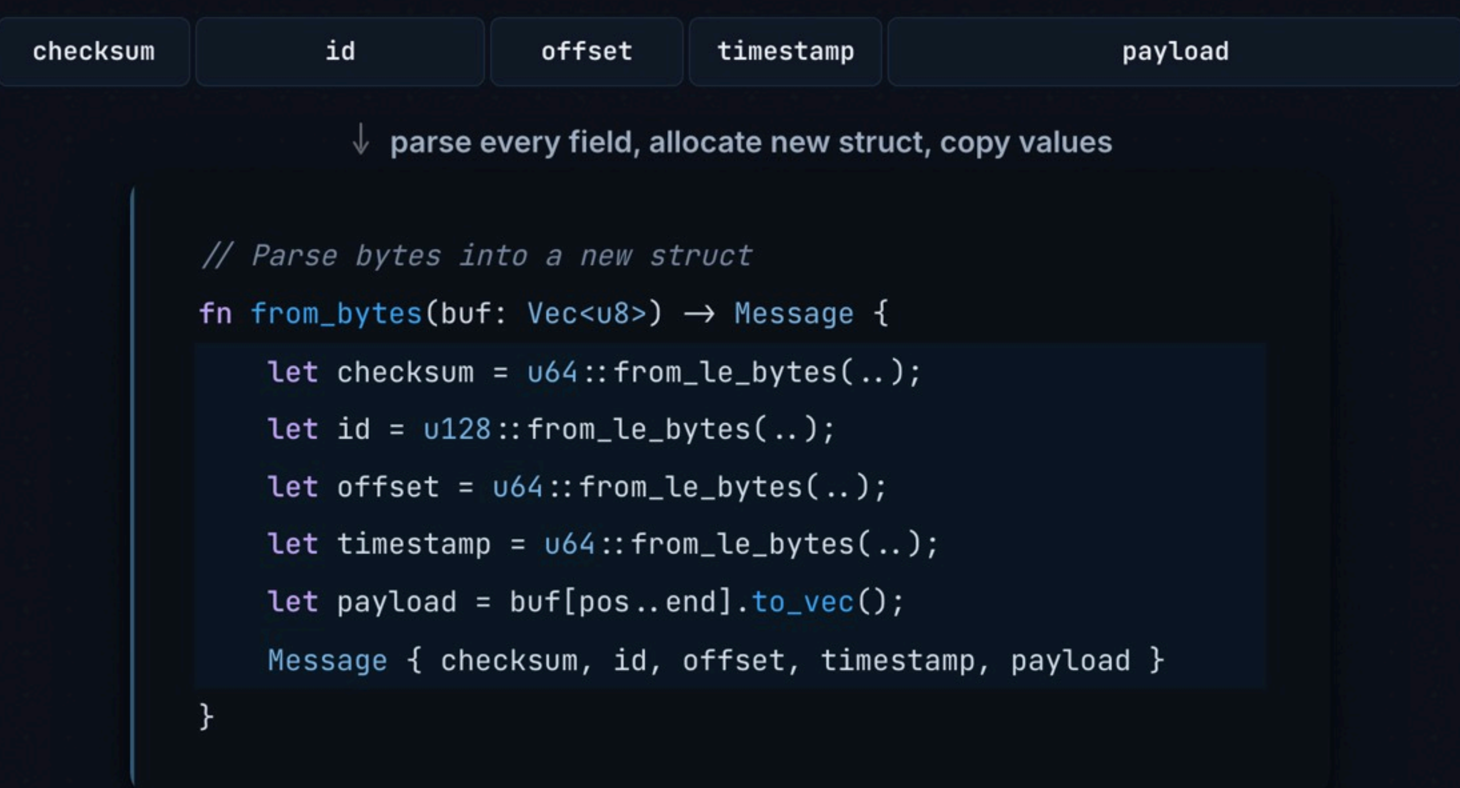


At p99: 10x worse, hundreds of ms lost



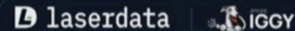
Where Do the Milliseconds Go?

The cost of copying data



New message constructed, payload copied, every message, every time

the hot copy here is payload.to_vec(): allocate a new Vec<u8> and copy payload bytes



The Zero-Copy Way

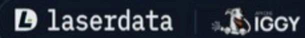
Serialization without allocation

Same buffer, zero copies

```rust
// Zero-copy: a view borrowing the wire bytes
struct IggyMessageView<'a> {
    buffer: &'a [u8],
}

// Fixed offsets in the 64-byte header
fn offset(&self) -> u64 {
    u64::from_le_bytes(&self.buffer[24..32])
}

fn payload(&self) -> &'a [u8] {
    &self.buffer[self.payload_offset..]
}
```

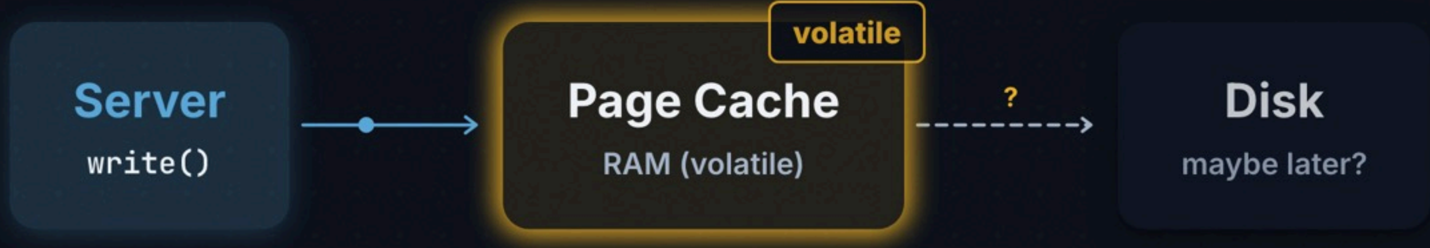
0 copies, 0 allocations

just a reference into the same buffer



But Wait - Is Our Data Safe?

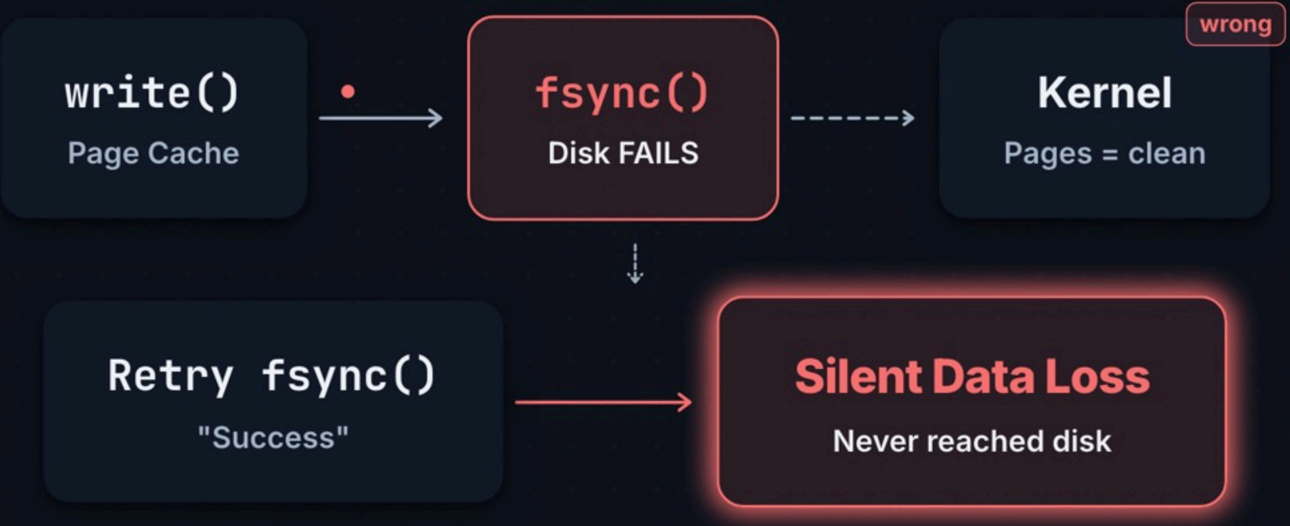
write() doesn't mean it's on disk



- write() lands in Page Cache (RAM), not on disk
- Kernel flushes to disk when it feels like it
- Crash or power loss = data gone

fsyncgate

Even fsync() can lie to you



- Failed fsync marks pages clean, retry silently returns success
- Your server thinks data is safe, but it was never written to disk
- What if we could bypass the page cache entirely?

The Fix - Direct I/O

Skip the page cache, write straight to disk

```
// O_DIRECT - bypass page cache
// optionally | O_DSYNC - durable on return
let file = OpenOptions::new()
    .write(true)
    .custom_flags(libc::O_DIRECT)
    .open(path).await?;

// Page-aligned buffer, recycled from the pool
let (buf, _) = memory_pool().acquire_buffer(len);
// fill buf with messages - business logic
file.write_all_at(buf, pos).await?;
```

- No fsyncgate, errors are immediate
- Aligned buffers pooled and reused, no per-write allocation
- Bypasses the page cache, no writeback jitter or double-buffering
- Alternative: `RWF_UNCACHED` (Linux 6.14), buffered write that drops the cache behind itself

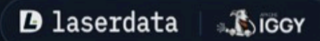
Stop Allocating on the Hot Path

A pool of page-aligned buffers, recycled

acquire → use → release   28 size classes · 4 KiB to 512 MiB

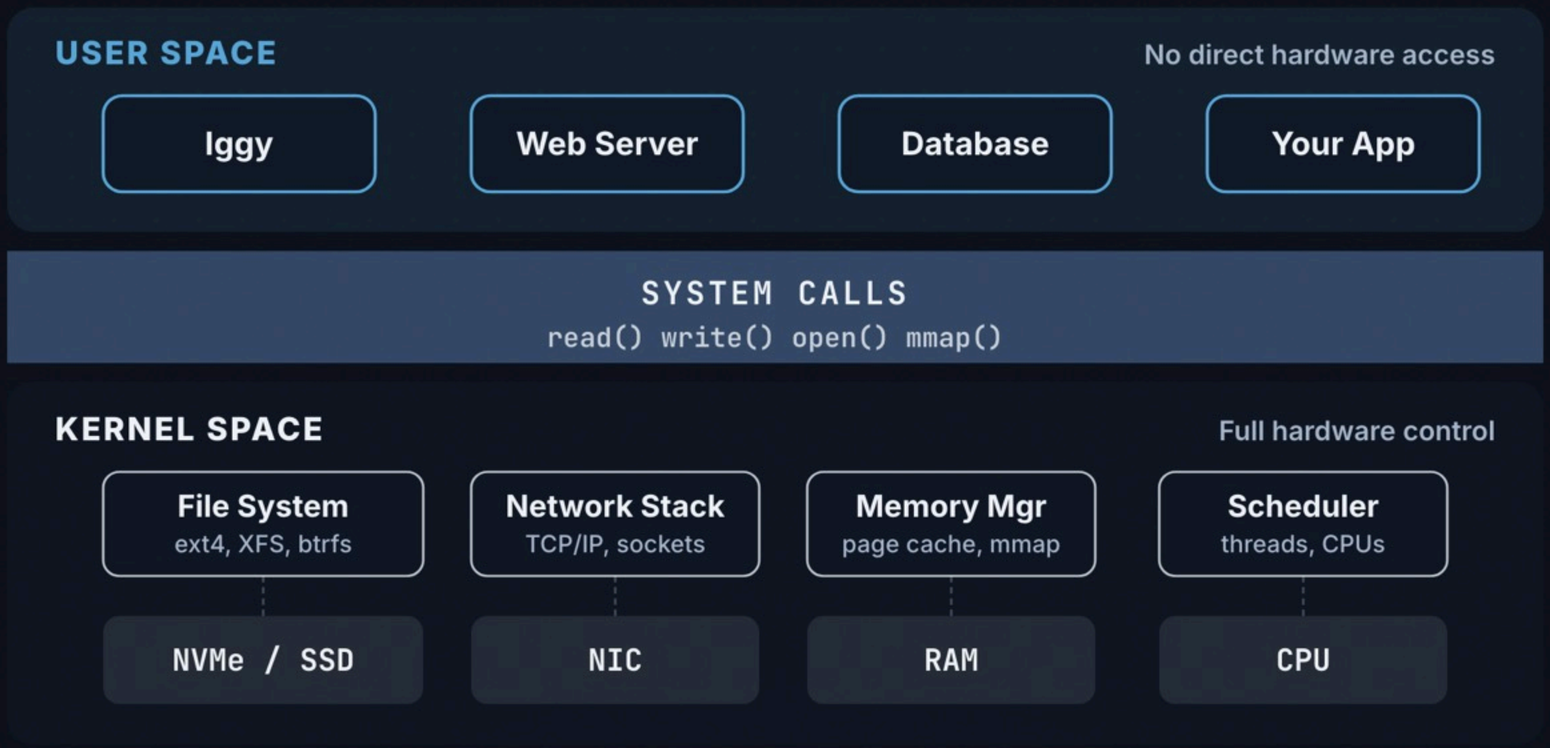
```
let (buf, pooled) = memory_pool().acquire_buffer(capacity);
// fill buf with messages - business logic
// optional: register pool once for io_uring fixed buffers
// write buf on the I/O path (O_DIRECT / io_uring)
buf.return_to_pool(original_capacity, pooled);
```

- Buckets by size class, each a lock-free queue of free buffers
- No allocator calls or first-touch faults after warm-up
- 4 KiB-aligned buffers can become `io_uring_register_buffers()` fixed buffers

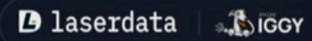


How Does Your App Talk to Hardware?

Every disk read, every network packet goes through the kernel



Syscalls can be expensive: a context switch makes the CPU synchronize its caches



How Do We Talk to the Kernel?

`epoll`: readiness-based I/O

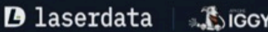
Network I/O — Thread is free while waiting

Thread: send | waiting (thread sleeps) | ready! | process — time →

Disk I/O — Thread stuck, waiting for disk

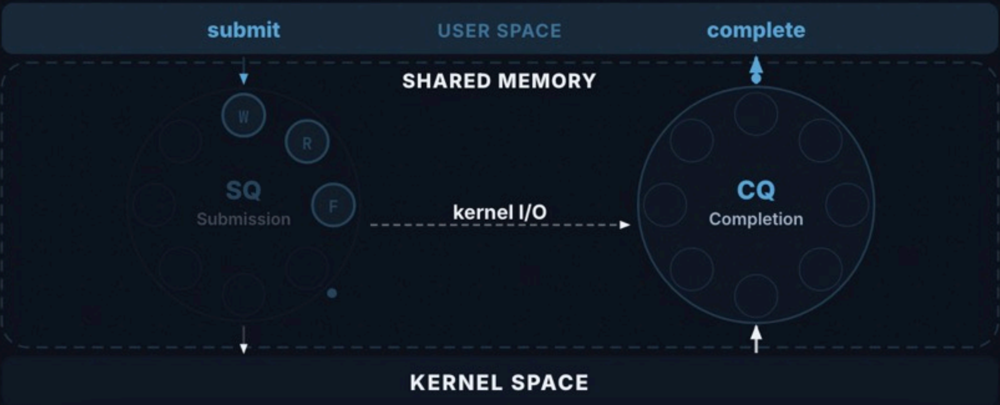
Thread: send | blocked (waiting for disk) | done — time →

- Great for network sockets, not designed for disk
- Disk reads block the thread until complete
- `io_uring` gives true async disk I/O



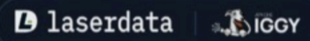
A Better Way: io_uring

Shared ring buffers, batched submission




```
async fn append<B: IoBuf>(buf: B) {
    let (result, _buf) = file
        .write_all_at(buf, pos).await;
}
```

- Batched via shared memory, one syscall per batch
- Fixed buffers: register pool once, submit buffer ids with `write_fixed`
- True async disk I/O, thread never blocks



Why Not Just Add io_uring to Tokio?

Readiness vs completion - a fundamental ownership mismatch

Tokio (epoll) `&mut buf - caller owns`

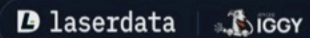
```
pub trait AsyncRead {
    fn poll_read(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
        buf: &mut ReadBuf<'_>,  // borrowed
    ) → Poll<io::Result<()>>;
}
```

Compio (io_uring) `buf: B - moves to kernel`

```
pub trait AsyncRead {
    async fn read<B: IoBufMut>(
        &mut self,
        buf: B,  // moves to kernel
    ) → BufResult<usize, B>;
}
```

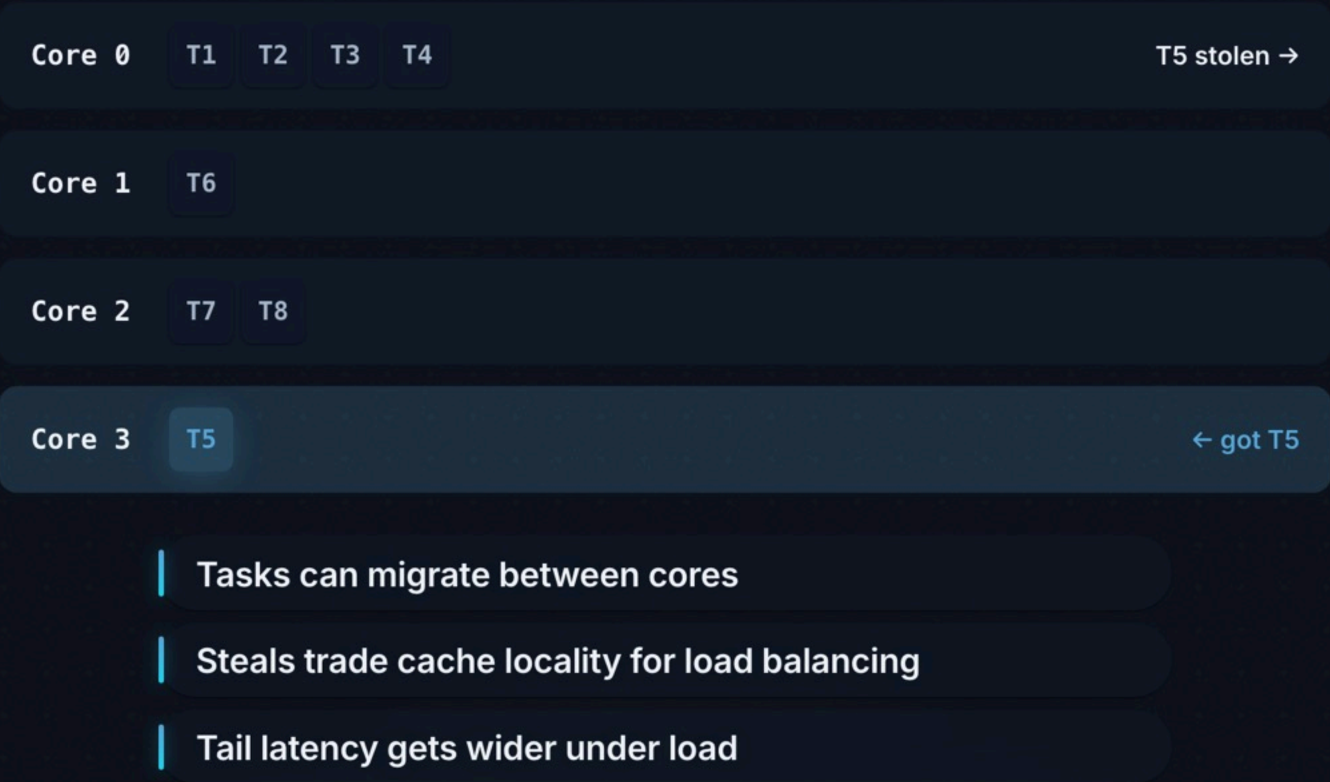
Runtime	Status	Notes
Monoio	EVALUATED	Limited io_uring API coverage, low maintenance activity
Glommio	EVALUATED	Largely stagnant, design philosophy disagreements
Compio	CHOSEN	Active maintenance, broad io_uring coverage, decoupled driver-executor

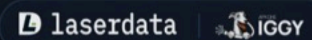
- Tokio assumes the caller owns the buffer
- Compio hands buffer ownership to the kernel, returned on completion
- Per-shard Compio runtime, `COOP_TASKRUN`, Linux kernel 5.19+



How Runtimes Use Your CPU

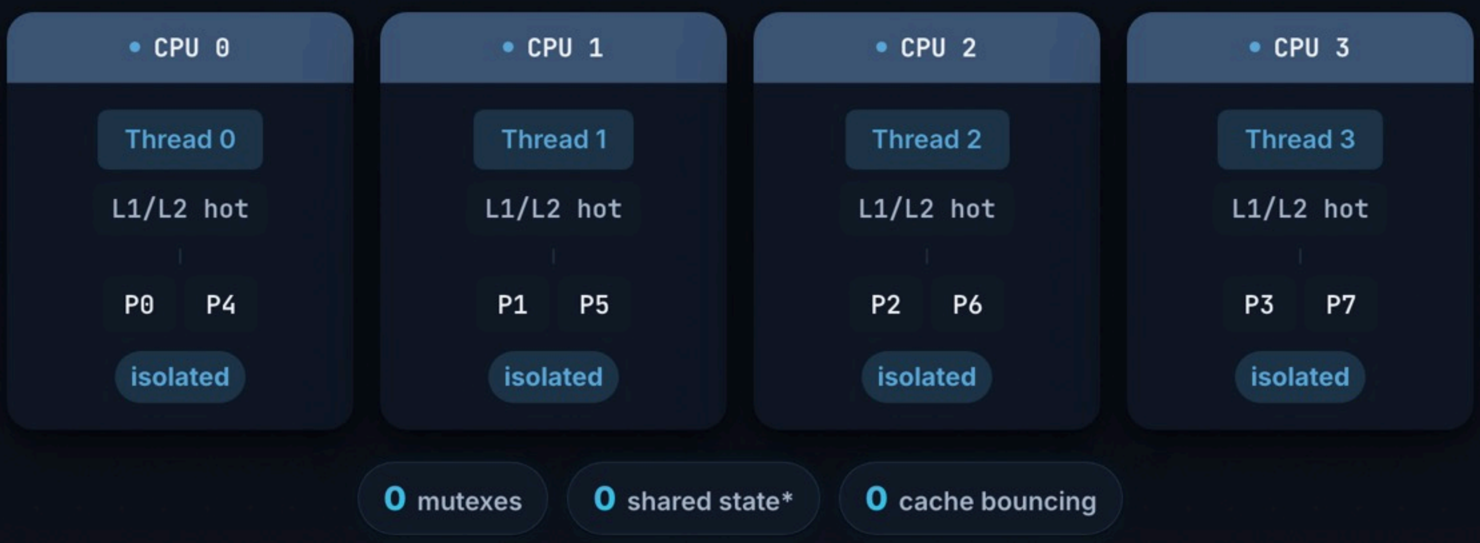
Work stealing - free workers grab tasks from busy ones





What If We Don't Share?

Thread-per-core: pin one thread to one CPU, own your data



0 mutexes | **0** shared state* | **0** cache bouncing

* shared state still exists, done differently: `arc-swap` / `left-right`, each shard reads its own copy

- No locks, no shared mutable state
- CPU cache stays hot, no cross-core invalidation
- Predictable, stable tail latency

Thread-per-Core

CPU affinity

```rust
// Pin current thread to one CPU core
pub fn bind_current_thread_to_cpu(
    cpu: usize,
) -> nix::Result<()> {
    let mut cpuset = CpuSet::new();
    cpuset.set(cpu)?;
    sched_setaffinity(Pid::from_raw(0), &cpuset)
}
```

Shared-nothing partition

```rust
// Per-shard - single-threaded, NO locks
pub struct LocalPartition {
    pub log: SegmentedLog,
    pub offset: Arc<AtomicU64>,
    pub consumer_offsets: ConsumerOffsets,
    pub stats: PartitionStats,
}
```

Okay, We Need to Share a Little

Not everything can be sharded

```rust
fn dispatch(&self, msg: Message) {
    let shard = match msg.kind() {
        Metadata => 0,
        Partition => self.table.shard_for(msg.namespace()),
    };
    self.senders[shard].send(msg);
}
```

- Shard 0 owns metadata, every shard reads a synchronized copy via `publish()`
- Partition = unit of work, owned by one shard, work fans out over `MPSC` channels
- Zero locks on the hot path, only message passing

Compiles, Then Panics

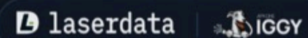
The `RefCell` trap, and what finally worked

THE FIX **left-right for metadata**

```
struct Server {
    streams: WriteHandle<Streams, Op>,
}
fn save_stream(&mut self, id: u64) {
    self.streams.append(Op::Upsert { id });
    self.streams.publish();  // atomic swap
}
// in-memory apply, no .await
```

✓ Zero locks on read path

- Readers use one copy, writer updates other
- `publish()` atomically flips, `arc-swap` works too for whole-map snapshots

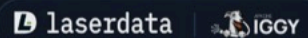


Know Your Hardware: NUMA

Pin each shard to a core, keep its memory on the same socket

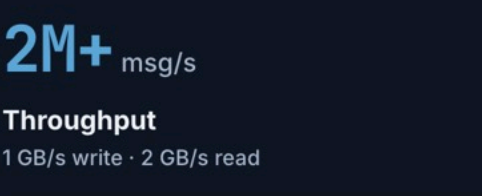
```
# config.toml
[sharding]
cpu_allocation = "numa:nodes=0,1;cores=4;no_ht=true"
```

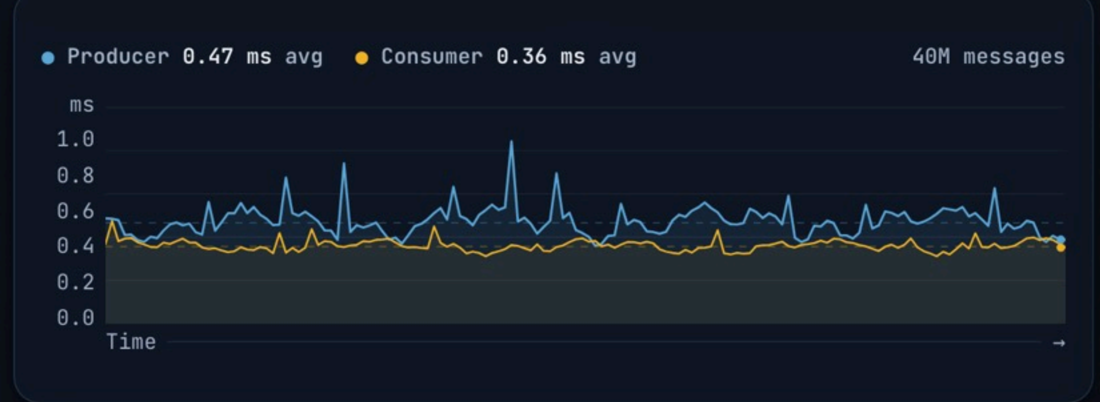
- `hwlocality` maps sockets, cores and threads at boot
- Thread-per-core pins the core, NUMA pins the memory
- Each shard allocates on its local node, no cross-socket hops



So How Fast Is It?

Benchmarked, not estimated - real Apache Iggy numbers





Latency breakdown (ms)

	Prod	Cons
Avg	0.466	0.357
Median	0.349	0.351
P95	0.886	0.446
P99	0.976	0.495
P99.9	1.114	0.566

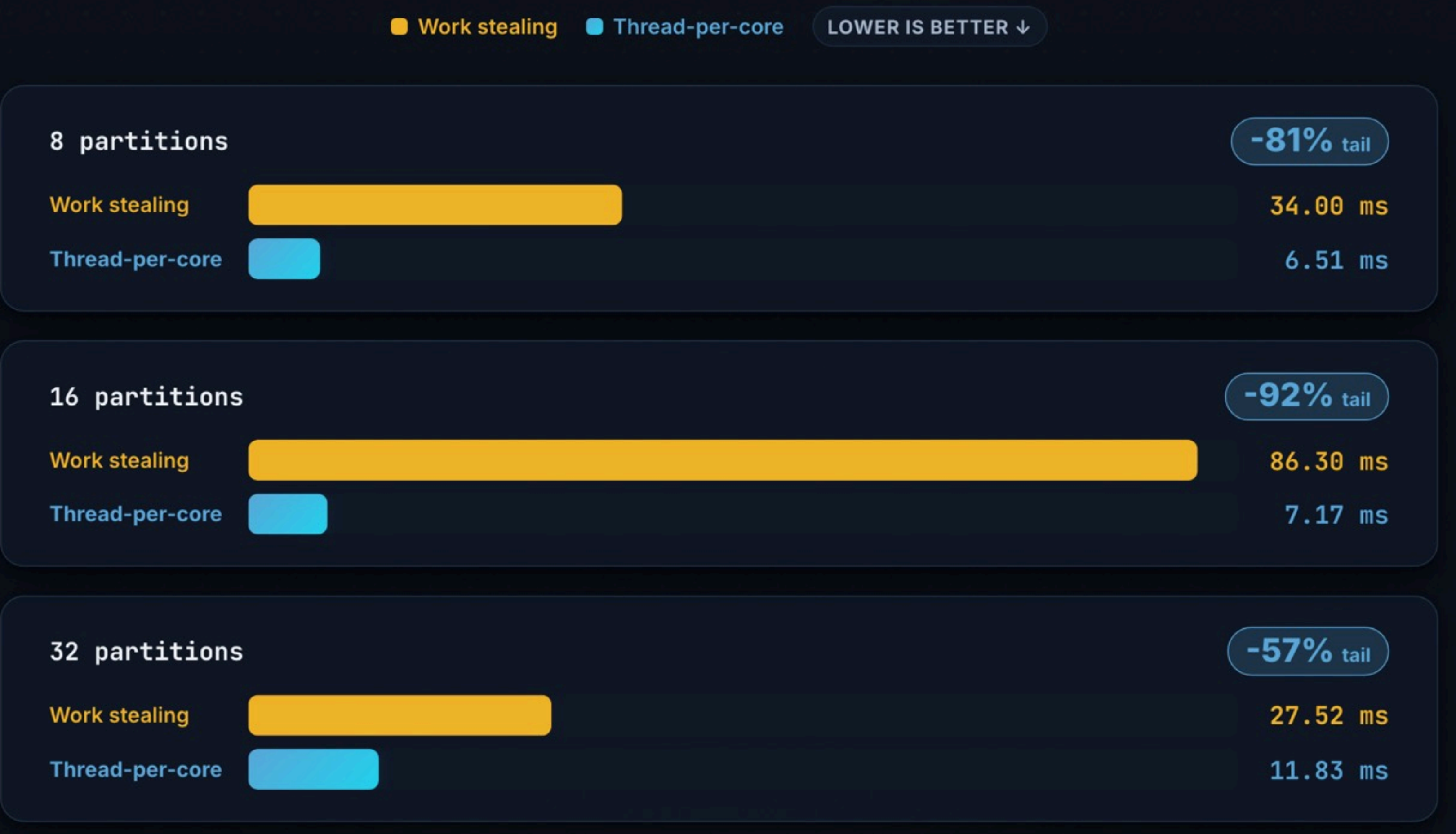
Machine: AWS i4i.4xlarge · Apache Iggy 0.8.0

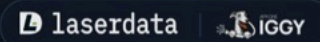
Workload: 16 producers · 16 consumers · 1000 B messages · 250 per batch

Producer → Consumer →

Before and After

Work stealing vs thread-per-core · P99.99 tail latency





Consensus in Existing Systems

Common replication and coordination patterns across current streaming systems

PLATFORM	CONSENSUS	NOTABLE COST
Apache Kafka	`KRaft (metadata)`	Metadata quorum, partitions use leader + ISR
Apache Pulsar	`BookKeeper + ZK`	BookKeeper for data, ZooKeeper for coordination
Redpanda	`Raft`	Per-partition Raft group
NATS JetStream	`Raft`	Per-stream and per-consumer Raft groups
RabbitMQ Streams	`Raft`	Per-stream leader + replicas

COMMON OPERATIONAL PATTERNS

- Leader failover still depends on election and recovery paths
- Majority quorum still gates availability during failures

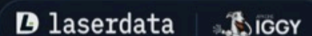
Iggy chose a different path: `Viewstamped Replication`

Viewstamped Replication Revisited

Deterministic primary · quorum commit · predictable failover

Primary replicates → backups acknowledge → primary commits

- Quorum durability: committed once a majority holds it, not a blocking fsync per write
- Predictable failover: deterministic primary by view number, no randomized timeouts
- Simpler leadership: view changes replace per-term vote bookkeeping



Down the Rabbit Hole

On the roadmap - there's always one more thing to optimize

Vectorized I/O

writev batching, fewer syscalls

io_uring Fixed Buffers

Pre-registered buffers, SQPOLL

NUMA-partitioned Pool

Per-node buffer arenas

VSR Clustering

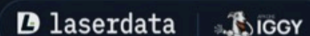
Multi-node replication, landing in server-ng

kTLS

Kernel-offloaded encryption

Kernel Bypass

DPDK, XDP: skip the kernel



Why Are We Doing This?

Modern hardware deserves modern software

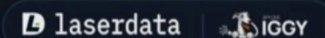
Sub-ms Latency Millions of msg/s	**SDKs** Rust, Go, Java, C#, Python	**Connectors** InfluxDB, Delta Lake, Doris	**Web UI & CLI** Dashboard, benchmarks
Transports TCP, QUIC, WS, HTTP	**Observability** OTel, Prometheus	**Clustering** VSR, coming soon	**Apache 2.0** Open source, forever

2023-04 — 2026-06

`5159221` 2026-06-11 `feat(server-ng): add partition reconciliation loop`

• last commit 6d 20h 8m ago

1,900+ commits	**100+** contributors	**4,300+** GitHub stars	**Apache** Software Foundation



Built by the Community

Contributors from around the world, building next-gen infrastructure in the open

Tag	#	Title	Contributor
cluster	#3269	Multi-shard cross-shard messaging	Hubert Gruszecki
VSR	#3254	VSR header framing	Grzegorz Koszyk
Java SDK	#3026	Java client connection timeout	Jonathon Henderson
metadata	#2916	Persistent metadata WAL journal	Krishna Vishal
PHP SDK	#3235	Initial PHP SDK	Diaconu Radu-Mihai
C# SDK	#3250	Rented zero-alloc message polling	Łukasz Zborek
Go SDK	#2964	context.Context on client methods	Chengxi Luo
C++ SDK	#3046	Messaging FFI functions	xin
connector	#3140	InfluxDB v2 & v3 connector	ryerraguntla
connector	#2889	Delta Lake sink connector	Kriti Kathuria
server	#3233	systemd watchdog integration	Jonathan Davies
CLI	#3096	CLI context & session status	Atharva Lade

Thank you!

laserdata

spetz

spetzu

iggy.apache.org

laserdata.com