Building Apache Iggy: How Rust powers ultra-low tail latencies with io_uring

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

Krishna Vishal & Kranti Parisa | laserdata | APACHE IGGY

Hello, Bengaluru!

Krishna Vishal

laserdata | APACHE IGGY

krishvishal

Kranti Parisa

laserdata | APACHE IGGY PPMC

kparisa

Every Millisecond Counts.

You're running a streaming engine. Throughput is fine.
Then you check your **P99 tail latencies**.

p50: 8.5ms / p99: 101ms

The streaming layer appears twice. A spike there doubles.

Streaming Isn't New!

Production-grade, widely adopted engines power today's systems

Apache Kafka
Distributed event log

Apache Pulsar
Multi-tenant stream platform

Redpanda
Kafka API compatible streaming

RabbitMQ
Queue-first broker

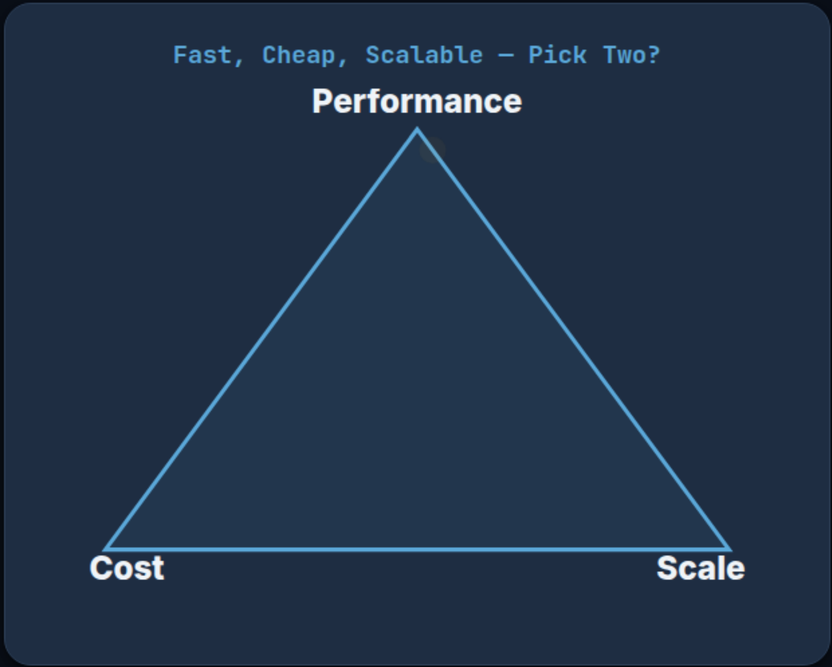
NATS
Cloud-native messaging

Amazon Kinesis
Managed stream service

Battle-tested systems with very real trade-offs

Great Systems, Hard Trade-offs

Performance vs Cost vs Complexity



- Tail latency spikes under mixed workloads
- Infrastructure cost rises quickly at scale
- Operational complexity compounds over time

Improving one side often increases pressure on the other two

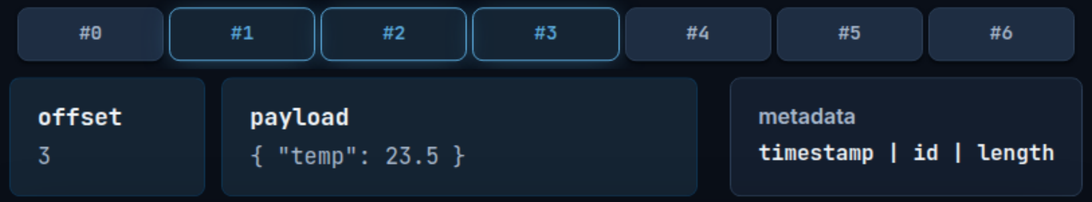
Why Another Engine?

Kafka was designed over a decade ago. Hardware has changed.

	2011	2026
STORAGE	HDDs (10ms seek)	NVMe (0.02-0.1ms)
CPU	4-8 cores	64-128 cores
I/O MODEL	select / epoll	io_uring
RUNTIME	JVM + GC pauses	Rust + zero-cost

What if we built from scratch for today's hardware?

Apache Iggy Single binary. Append-only log. Built entirely in Rust.




```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]
}
```

Putting It All Together

Stream / Topic / Partition / Segment

STREAM telemetry-services

TOPIC sensors

P0	seg_0	seg_1	seg_2	active	.log + .index
P1	seg_0	seg_1	active		
P2	seg_0	active			

partition_0/

seg_000.log	256 MB	
seg_001.log	256 MB	
seg_002.log	48 MB	active

seg_002.index

OFFSET	BYTE POSITION
0	0x0000
1	0x00A4
2	0x0148
3	0x01EC

O(1) offset lookup

Making It Production Ready

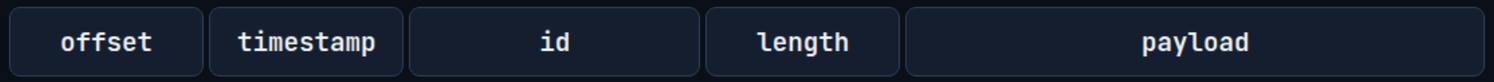
A working log is just the beginning.

1. How do we serialize without copying?
2. How do we guarantee data is actually on disk?
3. How do we make I/O truly async?
4. How do we use CPUs efficiently?

1. How do we serialize without copying?

The cost of copying data

A message arrives as a buffer



↓ parse every field, allocate new struct, copy values

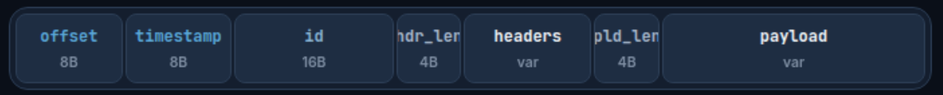
```rust
// Parse bytes into a new struct
fn from_bytes(buf: Vec<u8>) -> Message {
    let offset = u64::from_le_bytes(..);
    let timestamp = u64::from_le_bytes(..);
    let id = u128::from_le_bytes(..);
    let payload = buf[pos..end].to_vec();
    Message { offset, timestamp, id, payload }
}
```

Every field copied, new struct allocated, every message, every time

Iggy's Message Structures

The message is the bytes — no transformation, no owned struct

Wire format — bytes as they arrive from the network



IggyMessageView — points into the same bytes

```rust
// Just a reference — no allocation
pub struct IggyMessageView<'a> {
    buffer: &'a [u8],
}

fn offset(&self) -> u64 {
    u64::from_le_bytes(&self.buffer[0..8])
}
fn payload(&self) -> &'a [u8] {
    &self.buffer[payload_start..]
}
```

IggyMessageBatch — flat bytes, no per-message alloc

```rust
pub struct IggyMessageBatch {
    pub start_offset: u64,
    pub end_offset: u64,
    pub start_timestamp: u64,
    pub end_timestamp: u64,
    pub indexes: Vec<u8>, // u32 positions
    pub messages: Vec<u8>, // flat bytes
}
```

No owned IggyMessage in the hot path — just IggyMessageView borrowing the wire bytes

Batch stores indexes + flat messages as Vec<u8> — iterate by yielding views on demand

0 copies, 0 allocations just a reference into the same buffer

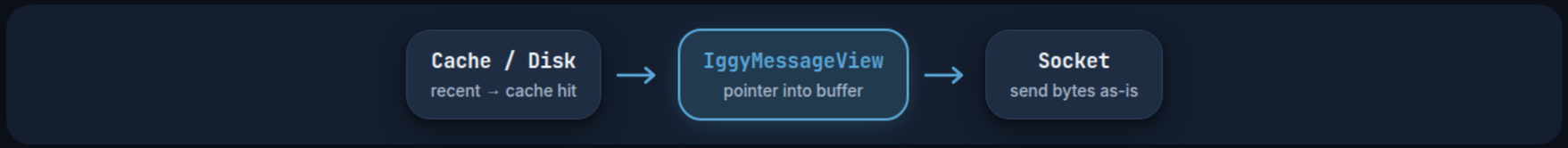
Zero-Copy Data Flow

Bytes flow through the system without transformation

WRITE PATH

Bytes from the network are appended directly — cached in memory and flushed to disk

READ PATH



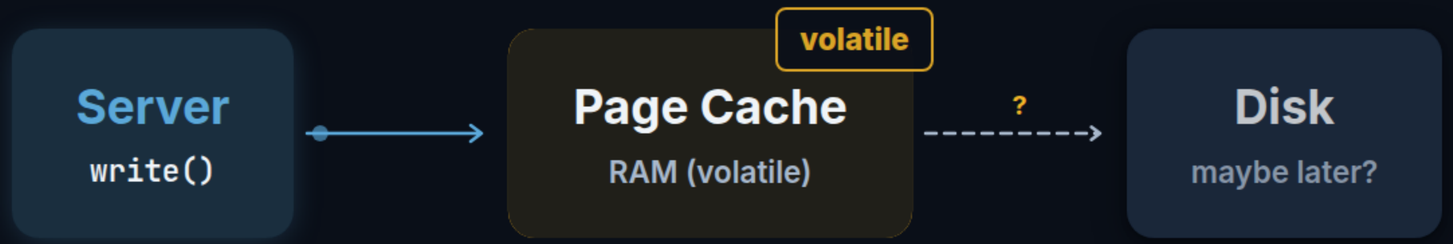
Recently appended messages are served from cache — no disk round-trip for hot reads

Zero copies, end to end

Same bytes on the wire, in cache, on disk, and back to the client

2. How do we guarantee data is actually on disk?

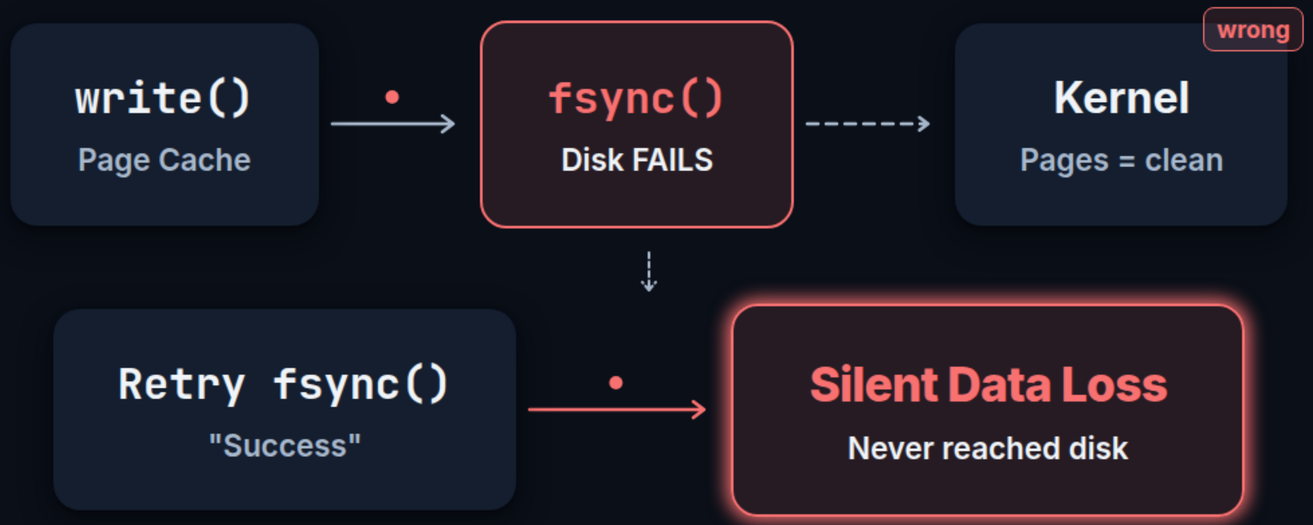
`write()` doesn't mean it's on disk



- 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

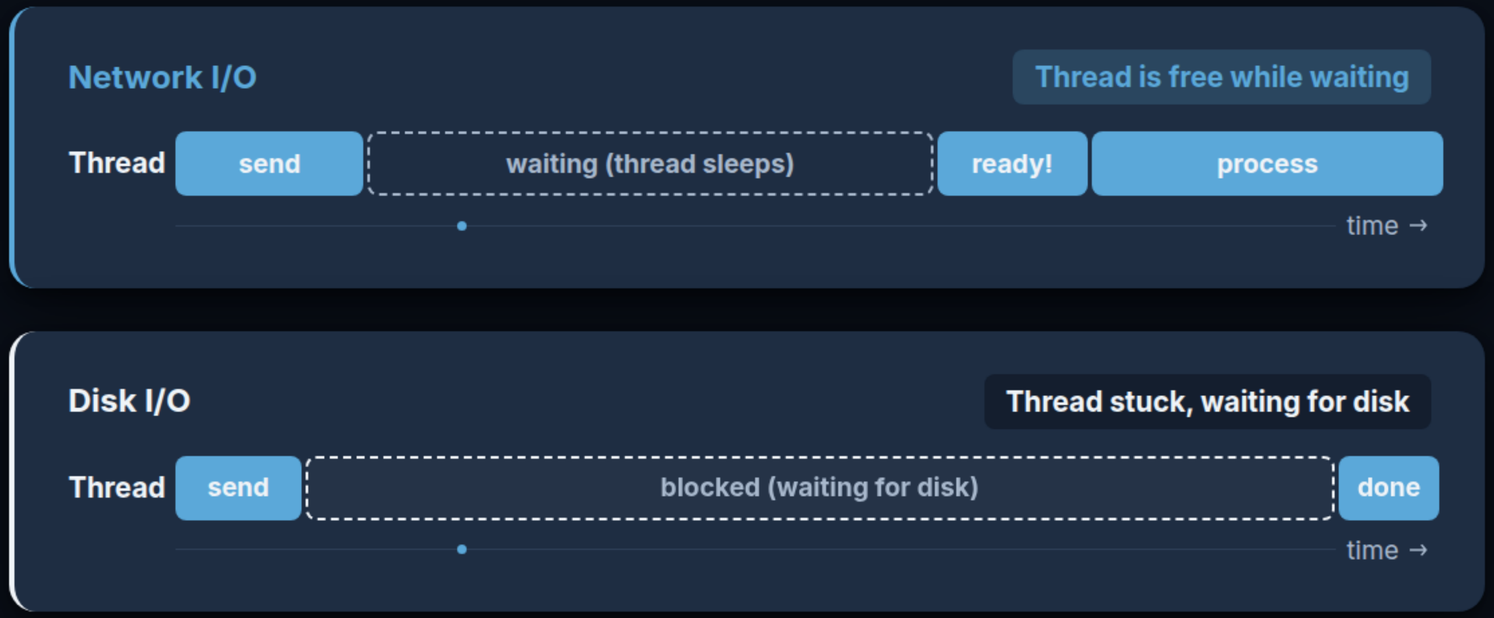

```rust
// O_DIRECT - bypass page cache
let file = OpenOptions::new()
    .write(true)
    .custom_flags(libc::O_DIRECT)
    .open(path).await?;

// Aligned buffer - 4096-byte pages
let buf = AlignedVec::new(4096, data);
file.write_all(&buf).await?;
```

- No fsyncgate, errors are immediate
- Requires aligned buffers (4096-byte pages)
- Data hits disk, no kernel caching in between

3. How do we make I/O truly async?

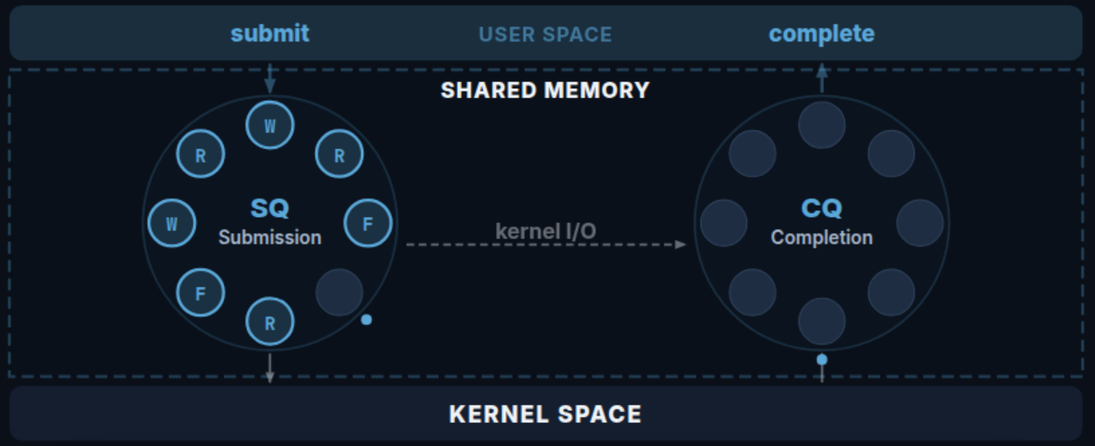
Tokio uses epoll — great for sockets, but disk I/O syscalls block the OS thread



- epoll notifies when a socket is ready — CPU sleeps, no wasted cycles
- Disk has no readiness concept — read()/write() block until the kernel finishes
- Tokio spawns blocking threads as a workaround, but that burns CPU and adds latency

A Better Way: io_uring

Shared ring buffers, zero syscalls per I/O




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

- Batched via shared memory, no syscall per I/O
- Buffer ownership passed to kernel, returned on completion
- But how does this work with Rust's async ecosystem?

Why Not Just Add io_uring to Tokio?

Readiness vs completion — a fundamental ownership mismatch

Tokio (epoll) — &mut buf — caller owns

```rust
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

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

Monoio — EVALUATED	**Glommio** — EVALUATED	**Compio** — CHOSEN
Limited io_uring API coverage, low maintenance activity	Became unmaintained, design philosophy disagreements	Active maintenance, broad io_uring coverage, decoupled driver-executor

> **Tokio's AsyncRead/AsyncWrite traits assume the caller always owns the buffer — incompatible with io_uring**

> **Compio uses IoBuf (writes) and IoBufMut (reads): buffer ownership transfers to kernel on submit, returned on completion**

> **Decoupled driver-executor lets Iggy plug in a custom thread-per-core executor**

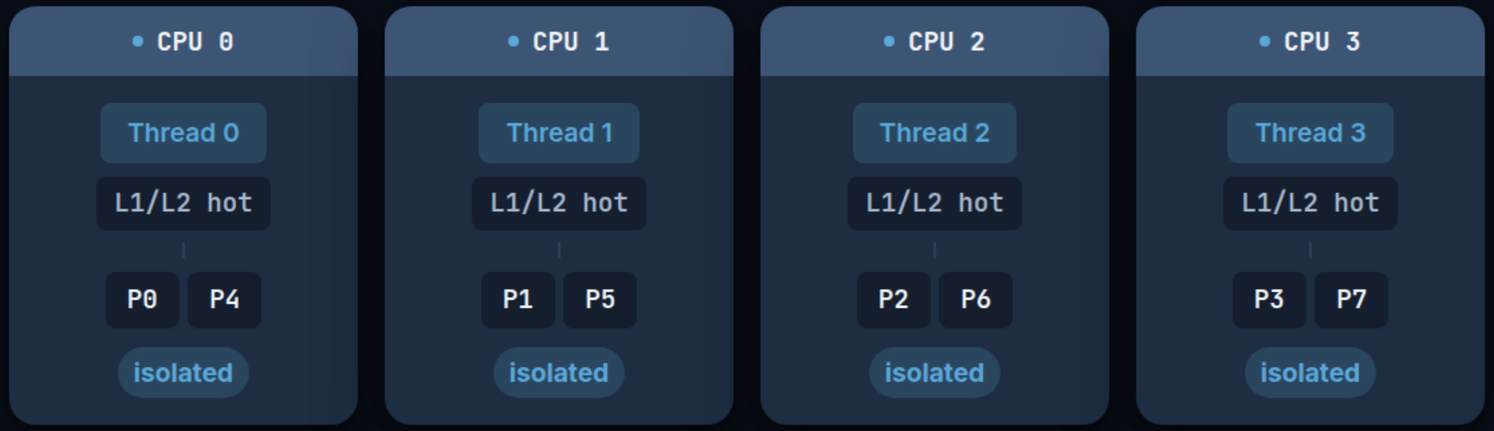
4. How do we use CPUs efficiently?

Work stealing - idle cores grab tasks from busy ones

- Tasks migrate between cores unpredictably
- Every steal invalidates CPU cache
- Tail latency spikes under load

What If We Don't Share?

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



0 mutex | **0** shared state | **0** cache invalidation

- 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 thread to CPU core
pub fn bind_cpu(&self) -> Result<()> {
    let mut cpuset = CpuSet::new();
    for &cpu in &self.cpu_set {
        cpuset.set(cpu)?;
    }
    sched_setaffinity(Pid::this(), &cpuset)?;
}
```

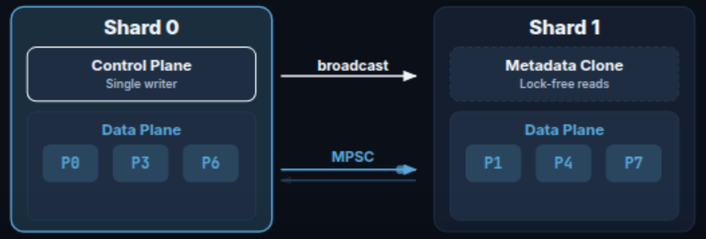
Shared-nothing partition

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

No `Arc<Mutex<...>>` in sight

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, broadcasts to all shards
- Partitions sharded across cores, MPSC channels for routing
- Zero locks on the hot path, only message passing

Compiles, Then Panics

The `RefCell` trap, and what finally worked

THE TRAP **RefCell across .await**

```
struct Server {
    streams: RefCell<Vec<Stream>>,
}

async fn save_stream(&self, id: u64) {
    let mut s = self.streams.borrow_mut();
    let stream = s.iter_mut()
        .find(|x| x.id == id).unwrap();
    stream.save().await;  // borrow alive here
}
// another task borrows → panic!
```

× Runtime panic

THE FIX **left-right for metadata**

```
struct Server {
    streams: WriteHandle<Streams, Op>,
}

fn save_stream(&mut self, id: u64) {
    self.streams.append(Op::Save { id });
    self.streams.publish();  // atomic swap
}
// no .await — metadata stays in-memory
```

✓ Zero locks on read path

- Two copies of the map — writers touch one, readers see the other
- publish() atomically swaps the pointer — readers never see torn state
- Lock-free reads, single writer — zero contention on the hot path

Metadata coordinated. Hot path stays shared-nothing.

So How Fast Is It?

AWS i3en.3xlarge · Intel Xeon 8259CL @ 2.50 GHz

1M+	+1 GB/s	+3 GB/s	2.05 ms
Messages / sec	Write throughput	Read throughput	P99+ latency

Work stealing vs thread-per-core — P9999 tail latency

benchmarks.iggy.apache.org

Consensus: The Status Quo

Most systems standardized on leader-based consensus — with real trade-offs

PLATFORM	CONSENSUS	NOTABLE COST
Apache Kafka	`KRaft (Raft)`	Per-partition leader
Apache Pulsar	`ZooKeeper (Zab)`	Separate ZK coordination tier
Redpanda	`Raft`	Per-partition Raft group
NATS JetStream	`Raft`	Per-stream Raft group
RabbitMQ	`Raft (quorum queues)`	Per-queue leader

SHARED ACROSS ALL OF THE ABOVE

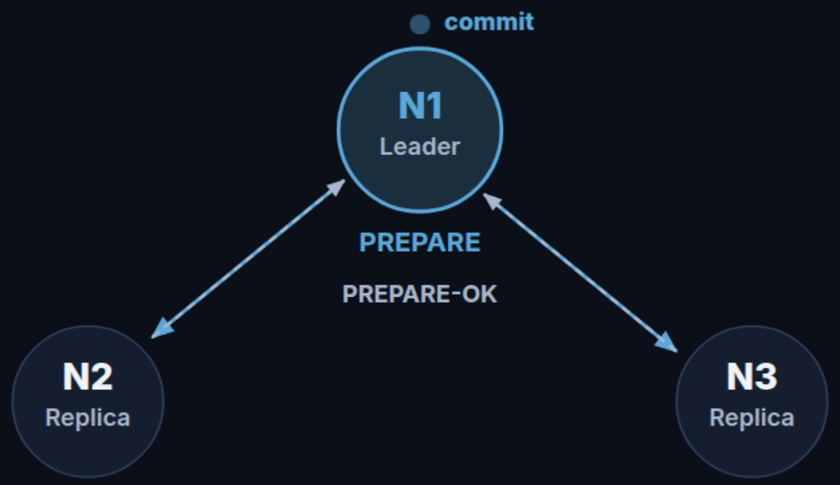
> **fsync on the commit path — durability tied to disk, not quorum**

> **Randomized election timeouts — failover latency variable by design**

Iggy chose a different path: Viewstamped Replication

Iggy's Answer: Viewstamped Replication

Work in progress, not yet in production



- Lower commit latency — quorum is the durability boundary; no fsync on the hot path
- Predictable failover — deterministic primary by view number; no randomized timeouts
- Simpler leadership model — view changes replace per-term vote bookkeeping

Chasing Microseconds

There's always one more thing to optimize

NUMA Awareness
Memory locality per socket

Memory Pool
32 buckets, 4 GiB pre-allocated

Vectorized I/O
writev batching, 1024 iovecs

Left-Right Metadata
Lock-free reads, atomic swap

kTLS
Kernel-offloaded encryption

Kernel Bypass
DPDK, XDP: skip kernel entirely

Why Are We Doing This?

Modern hardware deserves modern software

Sub-ms Latency
Millions of msg/s

SDKs
Rust, Go, Java, C#, Python, C++, Node

Connectors
Postgres, Mongo, ES, Quickwit, Iceberg, Stdout

Web UI & CLI
Dashboard, benchmarks

Security
TLS, AES-256, ACLs

Observability
OTel, Prometheus

Clustering
VSR, protocol-aware recovery, hash chaining

Apache 2.0
Open source, forever

2023-04 — 2026-04

```
44de98c  2025-02-14  refactor(server): improve cache size calculation
```

Built by the Community

io_uring #2299
Thread-per-core io_uring
@numinnex

NUMA #2412
NUMA awareness
@tungtose

VSR #2546
VSR view_change consensus
@krishvishal

shard #2476
Socket migration across shards
@tungtose

Java SDK #2606
Async connection pooling
@rythm-sachdeva

C++ SDK #2852
C++ low-level bindings & CI
@slbotbm

buffer #2944
TwoHalves buffer
@numinnex

zero-copy #2962
Zero-copy message primitives
@hubcio

Thank you!

Come find us at our table — let's deep-dive on your use case

laserdata

- laserdata.com
- laserdata
- laserdatainc

APACHE IGGY

- iggy.apache.org
- apache/iggy
- apache-iggy
- ApacheIggy
- apache-iggy