Drylane.io - The Dry Stack

Survival of the fittest, revisited. A working argument that optimization is good for the world.

TL;DR - don't read, click: live market wall, F1 replay, break a cluster, live benchmarks. Real engines, real queries, live right now, not screenshots.

A technical overview for engineers, architects, and anyone deciding where compute should go next.

> **A database, a message queue, and a serializer, fused into one process and tuned to the SIMD instruction. Tens of millions of writes per second. Microsecond queries. One machine.**

In one paragraph

Most systems are three separate machines wearing a trench coat: a database to store data, a message queue to move it, and a serializer to pack it on the wire. Every seam between them is a copy, a serialization pass, and a network hop — overhead you pay forever, on every record. The Dry Stack fuses all three into one process and tunes the result down to the SIMD instruction. The payoff is not a rounding error: on a single machine it ingests **tens of millions of records per second** and answers queries in **microseconds**, matching or beating clustered, purpose-built systems that run on an order of magnitude more hardware. Everything in this document is running live at drylane.io — real engines serving real queries, not screenshots.

The point isn't just that it's fast. The point is what "fast" *means*: the same work on far less hardware, far less energy, far less operational weight. Optimization is not a luxury or a hobby. It is the most direct lever we have on cost, on power draw, and on how good it feels to build software. This is the case for pulling that lever.

The problem: we've normalized over-provisioning

Somewhere along the way the industry decided that hardware is free and engineering time isn't, and then stopped checking whether that was still true. It quietly stopped being true years ago.

The default answer to "we have a lot of data and need it fast" is now reflexive: reach for a distributed database, put a message bus in front of it, wrap everything in containers, and schedule the containers across a cluster. Each layer is individually reasonable. Together they form a stack where the *majority* of the CPU cycles spent on a given request go to moving bytes between components that never needed to be separate — serializing, copying, hopping, deserializing, and doing it again on the way back.

We've learned to read the symptoms as the cost of doing business: the cloud bill that grows faster than the product, the cluster that needs a full-time team to keep upright, the "it's slow, add more nodes" reflex. But throwing nodes at a problem that's actually a *design* problem doesn't fix it — it rents around it, monthly, forever, and burns the electricity to match.

This document is not a complaint about any particular tool. Databases, brokers, and orchestrators are genuinely good at being general-purpose, and general-purpose is the right call for a great many teams. The argument here is narrower and more hopeful: **an enormous amount of that overhead is not fundamental.** It's there because nobody fused the pieces. When you do, the numbers move by factors, not percentages — and everything downstream of those numbers moves with them.

Optimization as a value, not a vanity

Three reasons this matters beyond the leaderboard.

Economic. If one box does the work of a cluster, the cluster's cost — machines, licenses, the people who babysit it — mostly evaporates. Efficiency is the only cost lever that doesn't ask you to ship less or charge more.

Environmental. Every unnecessary copy is an electron pushed for no reason. Data centers are a fast-growing slice of global electricity demand, and the marginal watt increasingly comes from wherever the grid can find it. Code that does the same job on a tenth of the hardware draws roughly a tenth of the power and needs roughly a tenth of the machines built, shipped, cooled, and eventually landfilled. Efficiency is climate policy you can merge.

Experiential. Fast is fun. Systems that answer before you've finished asking change how you think — you explore instead of wait, you ask the expensive question because it's no longer expensive. A generation of engineers has been taught that "good enough" is the ceiling and that latency is someone else's SLA. It isn't, and it doesn't have to be. Aiming higher is not nostalgia for hand-tuned assembly; it's a bet that the world is better when the tools underneath it are excellent.

The Dry Stack is the existence proof. Here's what's in it.

What the Dry Stack is

Three tightly integrated components, one process, no glue.

Component	Replaces	What it does
Facts	the database	Embedded time-series store. SIMD query surfaces over billions of records, answers in microseconds.
Nio	the message queue	Zero-copy async I/O and messaging. Tens of millions of msg/s on one connection, hundreds of millions fanned out.
Zao	the serializer	Tiered-codegen serializer whose fast path is a raw `memcpy` — the speed of memory itself.

The engine core is **C++**, kept lean and allocation-averse. On top sits an idiomatic **C#** layer with zero-copy wrappers: you write your application in C#, and the hot path pays no marshalling tax, no P/Invoke penalty, no per-record allocation. One codebase runs on **Windows and Linux**, tuned to the SIMD for **Intel, AMD, and ARM** — the same engine from a server rack down to a $35 Raspberry Pi.

Because the three share one address space, the seams disappear. A message doesn't get serialized, copied to a socket buffer, sent, received, copied again, and deserialized — it goes from buffer to wire and back. A stored fact doesn't cross a network to be queried — the query runs where the data already lives. Deleting the seams is where the order-of-magnitude comes from.

How it's actually fast

No single trick. A stack of deliberate choices that compound:

- **Fusion removes the hops.** Store, transport, and serialization live in one process, so the copies and network round-trips between them simply don't exist.
- **Zero-copy everywhere.** Buffers are pre-allocated and reused; data moves by reference, not by duplication. Nio's buffers are GC-free and identical whether you hold 64 connections or 10,000.
- **Columnar + SIMD.** Facts stores fixed-size fields column-by-column, so a SIMD kernel streams one column at full memory bandwidth instead of hauling whole rows through cache.
- **Blit serialization.** When a struct's in-memory layout already matches the wire, Zao's top tier just copies the bytes. No field walking, no encoding, no allocation.
- **Modern async I/O.** Nio sits on `io_uring` (Linux) and RIO (Windows), draining many connections with a handful of poll threads and no locks — instead of one thread per connection paying ~1 MB of stack each.
- **Saturate one box first.** The whole stack is designed to fill a single machine before it reaches for a second. Most workloads never need the second.

None of these are exotic in isolation. The result is unusual because they're applied *together*, to all three layers, at the same standard — so there's no slow component quietly capping the rest.

Proof, not slides

Every number below is a median from an open benchmark harness, on real data, on a single **AMD Ryzen 9 9950X** (16 cores / 32 threads). Each lives on a live page you can re-run.

Facts vs DuckDB — in-process, TSBS data

Same 8.19-million-row dataset, warm. Facts ran on **one core**; DuckDB got **all 32**.

Metric	Facts (1 core)	DuckDB (32 cores)	Gap
Ingest	20 M/s	2.3 M/s	**8.7× faster**
Latest-value point query	~0.2 µs	~4.3 ms	**~21,000× faster**
High-CPU filter (TSBS)	107 M/s	47 M/s	**2.3× faster**
Full-range aggregate	1.6 ms	7.9 ms	**4.9× faster**
Columnar scan (SIMD)	66 GB/s	7 GB/s	**9.4× faster**
Storage per point	96 B	25 B	3.8× *more* (honest loss)

The one column where Facts loses here is storage density — but that number needs an asterisk. Facts keeps *column* data raw and memory-mapped, on purpose: the hot columnar path stays uncompressed so SIMD kernels can scan it at full memory bandwidth. That's the trade the 96 B reflects, on TSBS's numeric, column-heavy data. *Blob* payloads are a different story: Facts compresses them through a codec you plug in via a simple function-pointer callback — the lowest-overhead extension seam there is — so you can bring the best compressor you have access to, and Facts stores the packed bytes and decompresses transparently on read. On blob-heavy data (logs, documents, packets), that can shrink the footprint substantially. It's a deliberate, *narrow* trade, not a blanket one. Details: drylane.io/facts.html.

Facts vs QuestDB & InfluxDB — over the wire

The real way you run those databases: client/server, every query crossing a connection and serialized back. Facts used a **single connection**; the rivals got **eight each** (their fastest setup — Facts doesn't get faster with more).

Metric	Facts (1 conn)	QuestDB (8)	InfluxDB (8)
Ingest	24.6 M/s	1.65 M/s (**15×**)	0.24 M/s (**102×**)
Latest-value point query	39 µs	260 µs (**6.7×**)	56 ms (**~1,400×**)
Range read	0.25 ms	1.15 ms (**4.6×**)	36 ms (**144×**)
High-CPU filter (TSBS)	5.2 ms	105 ms (**20×**)	846 ms (**163×**)

With full durability — every engine writing to a real NVMe SSD instead of RAM — Facts held at **25 M rows/s**; flushing to disk is essentially free for it. The gap is largely because Facts speaks a compact binary wire format while the rivals use text line protocols that re-parse every number as ASCII. (The rivals are also full query languages with far larger ecosystems — a real advantage this doc doesn't pretend away.)

Nio vs the field — messaging, loopback, 64-byte messages

Transport	Round-trip	Throughput	Bandwidth
Nio	16 µs	28 M msg/s	2.67 GiB/s
raw sockets	52 µs	28–40 M msg/s	2.34 GiB/s
NetCoreServer	33 µs	4.0 M msg/s	0.39 GiB/s
WebSockets	67 µs	75 K msg/s	0.86 GiB/s
gRPC	179 µs	49 K msg/s	0.15 GiB/s

Against the frameworks people actually reach for, Nio is 100–1000× ahead on round-trip. And where thread-per-connection collapses, Nio scales: **64 connections drained by 12 threads at 206 M msg/s** — about 7× the work per thread, with no locks. Details: drylane.io/nio.html.

Zao vs the field — encode into a reused buffer, nanoseconds (lower is better)

Payload	Zao	MemoryPack	MessagePack	Protobuf	JSON
PlayerMoved (16 B blit)	0.87 ns	3.11 ns	42 ns	141 ns	305 ns

Payload	Zao	MemoryPack	MessagePack	Protobuf	JSON
SpawnEntity (+ string)	12.5 ns	26 ns	58 ns	143 ns	282 ns
Snapshot (32 entities)	10.8 ns	36.7 ns	491 ns	1,997 ns	8,265 ns
Snapshot (1024 entities)	238 ns	335 ns	14,249 ns	63,875 ns	254,741 ns

Only MemoryPack stays in the same order of magnitude, and it still trails by 1.4–3.6×: the **fastest zero-allocation encode in .NET**. The honest caveat: Zao isn't the smallest on the wire — MessagePack's varint packing beats it on small, integer-heavy messages. Details: drylane.io/zao.html.

> **On methodology.** Rivals were given their best configuration — all cores, fastest bulk-load, parallel connections, reused buffers — and the places Drylane loses are shown in the same tables as the places it wins. The live benchmark also duty-cycles its ingest on purpose: run flat-out 24/7 it would write through a demo SSD in days, so it's capped to spare the disk. The speed is real; the cap just protects the hardware.

"Single box" — is that serious?

The most common objection, and a fair one. Single-box is the *fast path*, and for a surprising share of real workloads it's also the whole answer — but it isn't the only mode.

Facts includes epoch-based async replication with **veto-based failover**: a stale replica can't win an election, so two leaders can never exist. No split-brain, and a bounded, well-defined loss window set by a periodic checkpoint. Data is written in immutable, self-describing segments, each independently checksummed and recoverable, so a single bad block never takes the dataset down, and power loss doesn't corrupt the store. You can break a two-node cluster on purpose and watch it heal, live, at chaos.drylane.io.

For scale-out, a dataset splits into independent **shards** for parallel ingest and query — throughput scales with cores, no cross-shard coordination on the hot path. The design goal is to make one machine embarrassingly capable first, so that "add a node" is a real decision rather than a reflex.

Comfortable to build on

A fast engine you can't stand to use is a benchmark, not a foundation. The Dry Stack's application layer is C#, chosen on purpose: it's productive, it's memory-safe, and — importantly — it keeps getting faster **for free**. Microsoft has spent years relentlessly optimizing the .NET runtime and GC; every release, code sitting on top inherits work no single team could fund itself. Building the ergonomic layer there means the comfort improves over time without Drylane lifting a finger, while the C++ core holds the floor on raw speed. You write ordinary C#; the zero-copy wrappers make sure the hot path never pays for the convenience.

What's next (the TODO list)

None of the below is shipped. It's the roadmap, written as what it is — a list of things not done yet. The through-line is the same: take the optimization lens the stack already proves out, and point it at the next domain where "good enough" has quietly become the ceiling.

☐ **A new take on models that never stop learning** — a design in progress, built on foundations that already ingest and replay live data at these speeds.

☐ **Deployment via a launcher/updater** — ship and self-update these workloads the way games do, not the container-and-cluster default.

☐ **Audio processing** — real-time DSP on the same foundations.

☐ **Image processing** — GPU-accelerated (NVIDIA/CUDA) pipelines feeding the same storage and transport.

☐ Package the stack for general use (the open question behind the current launch: is there a NuGet the .NET world actually wants?).

These are directions, not promises. They're here because the thesis generalizes: most of the overhead we accept as fixed isn't.

Who's behind it

A solo systems engineer with 40+ years at the low level — from the Amiga demoscene and real-time audio DSP, through two decades of embedded payment and industrial systems, to a from-scratch maritime time-series database now being rolled out as the standard at a leading player in the marine industry. The Dry Stack is the stack I always wanted to build: fused, fast, and honest about its trade-offs. It exists because "good enough" got boring, and because Fast is Fun.

Are you building something that needs to be fast? Want to talk, build on it, or just kick the tires? Get in touch: **info@drylane.io**. More about the project at drylane.io/about.html.

See it all live: drylane.io, market wall, F1 replay, replication & failover, benchmarks, Raspberry Pi

---

Everything in this document is verifiable live at drylane.io. Figures are medians from open harnesses on an AMD Ryzen 9 9950X; each section links the page where you can re-run them. Share it, print it, feed it to your favorite model and interrogate it — the numbers hold up to scrutiny, which is the whole idea.