



CASE STUDY · DEEP DIVE EXHIBIT · ~6 MIN

WebAssembly in Salesforce LWC: 500K Record Engine & Net-Zero DB

“When the platform caps storage, heap, and CPU, stop fighting the caps - change where the data lives and where the work runs.”

ROLE

Independent Systems Researcher & Builder

TIMELINE

Independent research · 2025–2026

DOMAIN

Systems Engineering & High-Volume Web Architecture

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Independent weekend work on my own hardware, using synthetic catalog data I generated myself. The question: can a sandboxed CRM component (LWC) browse and search 500K+ catalog rows without stuffing rows into CRM storage, without dragging payloads through Apex heaps, and without freezing the tab? The approach I tested: compress a columnar JSON snapshot with gzip, stage it on object storage, stream it straight to the browser via short-lived signed URLs, decode with a Wasm-assisted path (Pako fallback), and search it with a small AST query engine over typed arrays - the same lineage as my open-source json-query-lite (TypeScript) and its Rust/Wasm sibling. User edits become small compressed delta files merged in memory at render time. No employer systems, no client data, no vendor internals - just the pattern, measured on my own fixtures.

Independent work, synthetic data only. I have watched large quoting and catalog problems hit the same three walls: CRM storage priced per row, server heaps capped in single-digit megabytes, and browsers asked to parse hundreds of megabytes of row-oriented JSON on the main thread. I wanted a reproducible answer to a narrow question - how far can you get with compression, direct-to-browser streaming, and in-memory indexing - so I built the pipeline end to end on weekends and measured it.

The walls, stated plainly:

- 1) Row storage is the wrong home: Half a million catalog rows do not belong in per-row CRM storage. The cost and the import time both hurt.
- 2) Server heaps are small on purpose: A 100MB+ JSON payload cannot pass through a 6–12MB execution heap. It fails loudly.
- 3) Upstream APIs throttle: Catalog providers rate-limit. Synchronous pulls during busy hours return 429s.
- 4) Fat JSON over the wire is slow: Row-oriented JSON with repeated keys compresses badly and parses slowly.
- 5) Main-thread parsing freezes tabs: Hundreds of thousands of object allocations trigger long GC pauses.
- 6) Edits seem to demand a database: Price overrides and margin tweaks look like they need RDS - with bills and ops to match.

CONSTRAINTS

- Stay inside the CRM component model: LWC under Lightning Web Security, no external portal redirect
- Cold open in tens of seconds, warm filtering instant: full load in under ~30s on broadband, queries in milliseconds
- No provisioned database for edits: overrides must work without RDS or similar
- No raw catalog rows in CRM storage: object storage holds the snapshot, CRM holds at most pointers
- Nothing heavy through the server heap: the browser streams bytes directly from storage

PART I · ARCHITECTURE

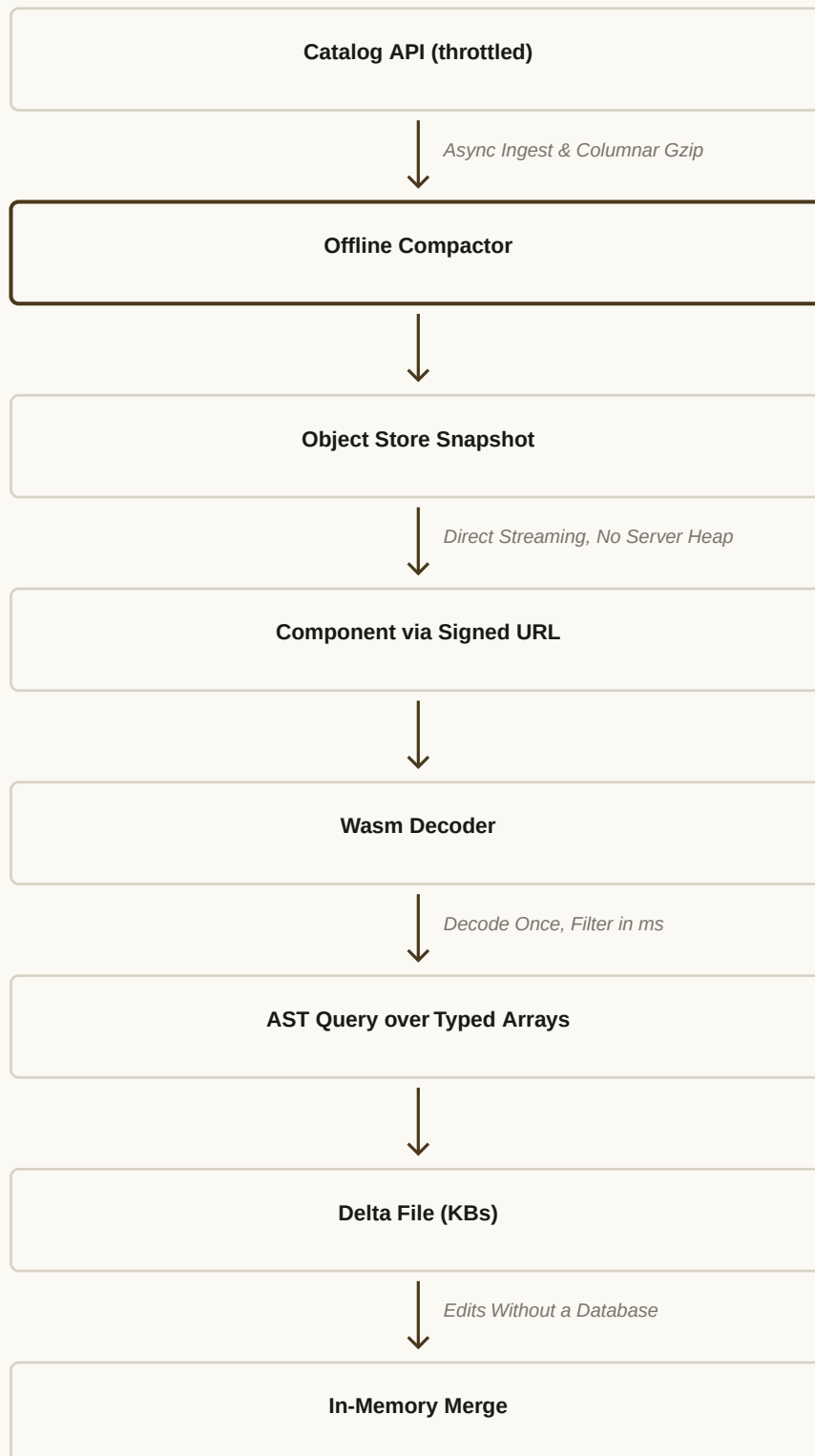
The system as designed

The pipeline I built and measured:

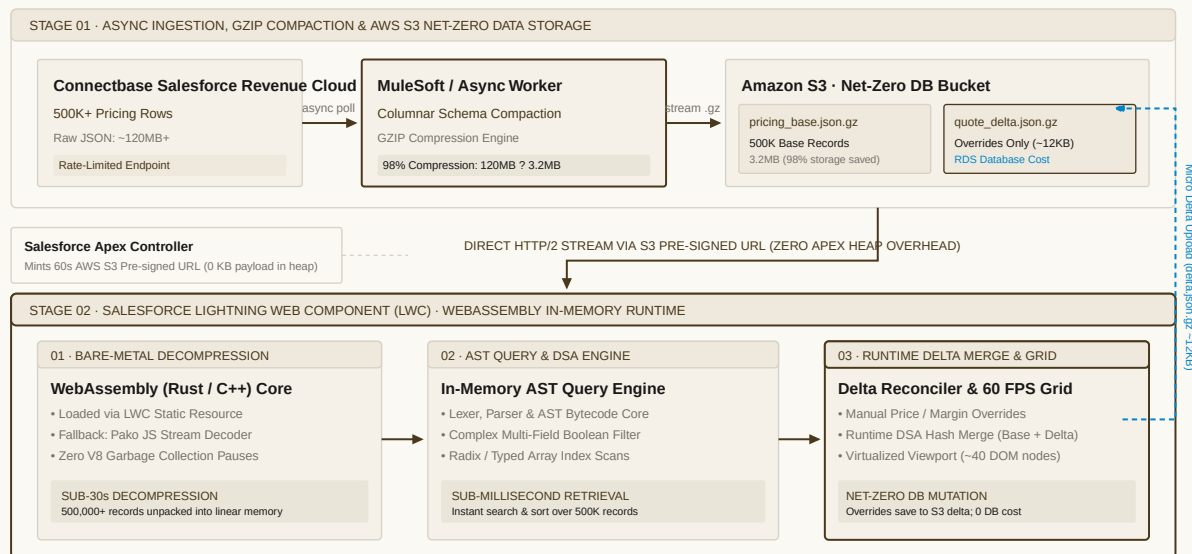
1. Offline compaction: a small worker reshapes synthetic catalog JSON into columnar arrays and gzip-compresses it. On my fixtures this lands around 2–3% of the original byte size, staged immutably on S3-compatible storage.
2. Signed-URL streaming: the server side mints a short-lived signed URL and steps aside. The component fetches bytes directly - the CRM heap never sees the payload.
3. Wasm-assisted decode with JS fallback: a small Rust/Wasm decoder expands the stream into linear memory; where Wasm is unavailable the same path falls back to Pako. Either way the result lands in typed arrays, not object soup.
4. In-memory query engine: the json-query-lite lineage - lexer, parser, AST over typed arrays and inverted indexes - evaluates nested AND/OR/NOT filters in milliseconds.
5. Delta files for edits: overrides serialize to a tiny compressed delta (~KBs) beside the base snapshot and merge in memory at load. No database process exists anywhere.
6. Virtualized grid: roughly four dozen DOM rows in a sliding window, so scrolling half a million logical rows stays smooth.

KULDEEP SINGH

HIGH-VOLUME CATALOG ENGINE & DELTA MERGE



Compressed snapshots stream direct to the browser, decode beside the GC heap, and merge with tiny edit deltas at render time.



Full system schematic - High-Volume Catalog Engine & Delta Merge.

PART I · DECISIONS

Choices, engineering, tradeoffs

Wasm-assisted decode vs. main-thread JSON parsing

Expanding a multi-megabyte gzip into hundreds of thousands of JS objects on the main thread means seconds of GC pauses.

CHOSEN

Small Rust/Wasm decoder into linear memory, with a Pako fallback on the same interface.

WHY

Decoding outside the GC heap kept the UI alive during ingest; the fallback keeps the same code path working where Wasm is restricted.

Direct signed-URL streaming vs. relaying through the server

Anything that touches the CRM heap inherits its limits.

CHOSEN

Short-lived signed URL; the component fetches bytes straight from object storage.

WHY

The payload never enters server memory. The server stays an auth gate, which is all it should be here.

Delta files vs. a provisioned database for edits

Overrides need durability, but not necessarily a running database.

CHOSEN

Immutable base snapshot plus small versioned delta files merged in memory.

WHY

For single-editor quoting sessions this removes the entire database tier. Concurrent editing needs explicit merge handling - stated openly below.

Typed-array AST engine vs. `Array.filter()` per keystroke

Naive per-keystroke scans across 500K rows lag Noticeably - a few hundred milliseconds per keypress.

CHOSEN

Small lexer/parser compiling filters to AST evaluated over typed arrays and inverted indexes.

WHY

The same engine family as my json-query-lite work. Millisecond queries without leaving the tab.

ENGINEERING NOTES

- Built the Wasm decoder from a minimal Rust crate (wasm-bindgen surface, streaming inflate) with an identical Pako JS fallback behind one interface, loaded as a static resource.
- Kept authorization server-side and narrow: short-expiry signed URLs scoped to one snapshot prefix; no credentials reach the client.
- Reshaped fixtures to columnar form before compression - shared keys factored out - which is where most of the byte savings actually come from.
- Reused the json-query-lite query core (lexer, recursive-descent parser, BM25-aware AST) over typed arrays; nested AND/OR/NOT plus LIKE/IN/BETWEEN work the same in Node and in the component.
- Implemented delta merge as a version-stamped hash join at load: base rows plus override rows reconcile before first paint, in milliseconds on test fixtures.
- Built the grid as a sliding window of a few dozen rows positioned with transforms, so scroll cost is constant regardless of row count.

PART I · OUTCOMES

Tradeoffs, results, lessons

Browser memory vs. server spend

Gain: No database or compute tier to operate for browsing and editing snapshots.

Cost: The tab holds the working set - on the order of 150–180MB for the largest fixture I tested.

Mitigation: Columnar typed arrays (strings as dictionary IDs, money as integer minor units) plus prompt release of parse buffers.

Wasm capability vs. sandbox reality

Gain: Near-native decode speed outside the GC heap.

Cost: Sandboxed environments restrict instantiation modes and shared memory.

Mitigation: Conservative Wasm feature set and a zero-config JS fallback; the slow path still works.

Delta simplicity vs. concurrent-write safety

Gain: No lock management, no connection pooling, full history for free.

Cost: Concurrent edits to one quote need optimistic concurrency, not row locks.

Mitigation: Version-stamped deltas with hash checks and last-writer-wins plus manual re-merge surfaced in the UI - documented as a limit, not hidden.

OUTCOMES

- 500K-row synthetic fixture cold-loads, decodes, and indexes in the high tens of seconds on broadband in my tests - dominated by download, not parsing
- Snapshot bytes shrink to roughly 2–3% of the row-oriented JSON on test fixtures via columnar layout plus gzip
- Interactive filtering over the full set returns in milliseconds per keystroke in the harness - no server involved
- Edits persist as KB-scale delta files with no database process to run or pay for
- Scrolling cost stays flat: constant DOM size regardless of logical row count
- Published lineage as open source: json-query-lite (TS) and a Rust/Wasm search sibling, both MIT

LESSONS

- Measure the heap you actually have before designing the pipeline - it decides everything downstream.
- Columnar layout plus gzip beats clever parsing of row-oriented JSON every time.
- Delta files are a legitimate database replacement for single-editor snapshots - and an honest limitation for concurrent ones.
- Direct-to-browser streaming removes a whole class of middle-tier bottlenecks.

01 · ORIENTATION

An exhibit, not an article

Seven layers, each with something to touch. Start anywhere - the path remembers where you are.

You are about to fit half a million rows through limits designed to stop you.

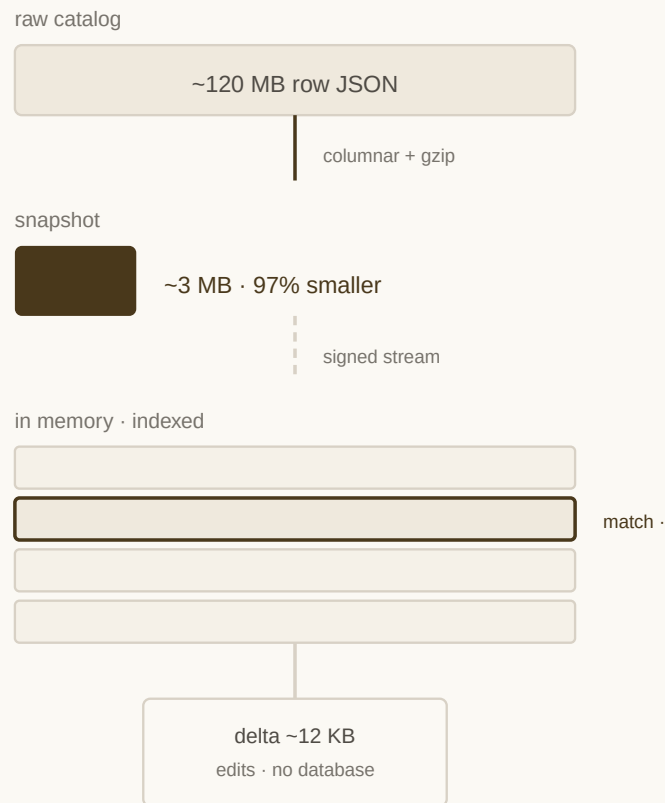


Exhibit plate - the system at a glance.

This exhibit traces independent systems research: a compressed catalog snapshot streams straight to the browser, a Wasm-assisted decoder expands it beside the garbage collector, an in-memory query engine answers in milliseconds, and edits persist as kilobyte deltas with no database anywhere.

All fixtures are synthetic and self-generated. What transfers to real work is the pattern: compress, stream past the server, decode beside the heap, index in memory, and merge deltas at render.

How to use this exhibit

- Inspect each stage of the ingestion-to-grid pipeline
- Step one snapshot from frozen bytes to searchable rows
- Decide what belongs on the server versus in the tab
- Break the pipeline at its thinnest points

02 · THE PROBLEM

Three load-bearing walls

Storage priced per row, heaps capped in megabytes, and a tab asked to swallow it all.

Three walls, all load-bearing: row storage, server heaps, and the main thread.

Half a million catalog rows do not belong in per-row CRM storage. A hundred-megabyte JSON payload cannot pass through a single-digit-megabyte execution heap. And parsing hundreds of thousands of objects on the main thread freezes the tab into surrender.

Edits seem to demand a database - with bills and operations to match. The exhibit exists to show which of these walls are real and which are routing problems.

THE CONSTRAINTS THIS FAILURE IMPOSES

- Stay inside the CRM component model: LWC under Lightning Web Security, no external portal redirect
- Cold open in tens of seconds, warm filtering instant: full load in under ~30s on broadband, queries in milliseconds
- No provisioned database for edits: overrides must work without RDS or similar
- No raw catalog rows in CRM storage: object storage holds the snapshot, CRM holds at most pointers
- Nothing heavy through the server heap: the browser streams bytes directly from storage

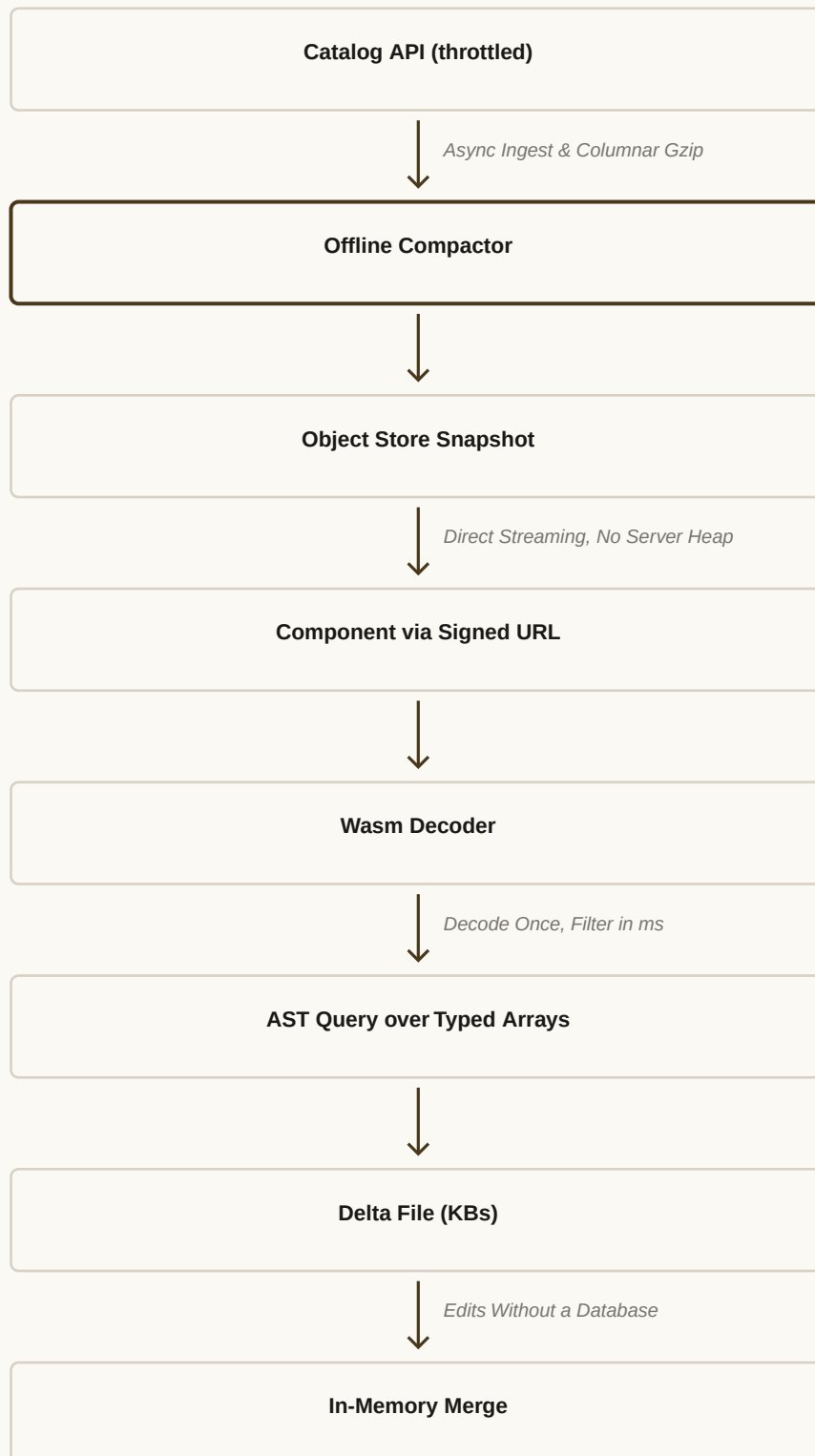
03 · THE ARCHITECTURE

Inspect the system

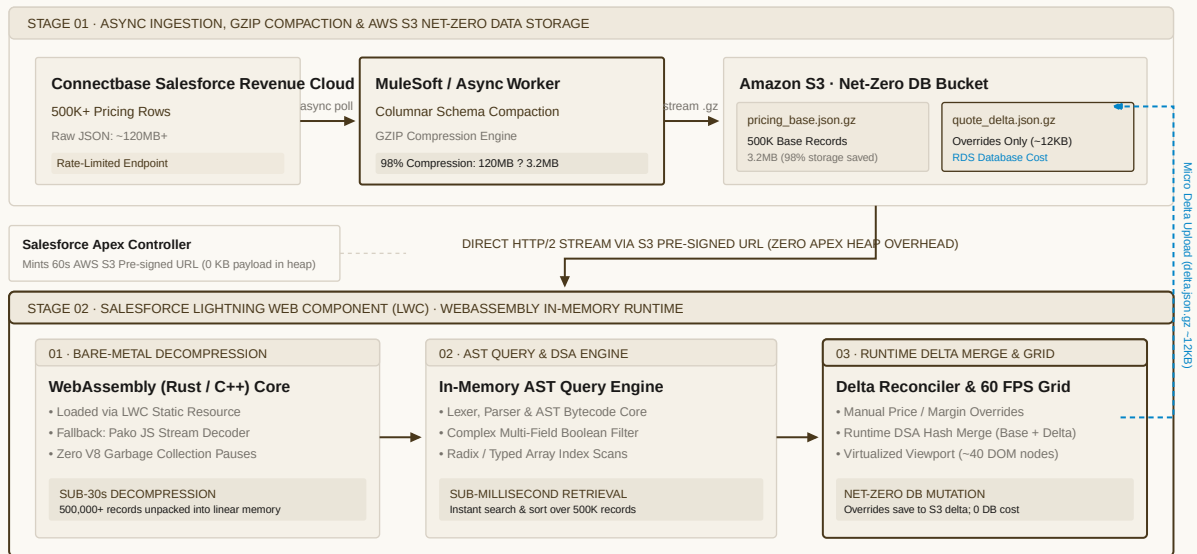
From frozen snapshot to searchable grid without touching a server heap.

Six stages. The server is an auth gate; everything else happens at the edges.

HIGH-VOLUME CATALOG ENGINE & DELTA MERGE



Compressed snapshots stream direct to the browser, decode beside the GC heap, and merge with tiny edit deltas at render time.



Full system schematic - High-Volume Catalog Engine & Delta Merge.

Boundary 01 / 06

Catalog Source

Row-oriented JSON in the wild.

PURPOSE

Represent the hostile starting point honestly: huge, repetitive, row-oriented JSON.

BOUNDARY

It is measured, never solved. Nobody edits the source format to flatter the pipeline.

FAILURE PREVENTED

Benchmark theater, where impressive ratios come from cooperative data.

Boundary 02 / 06

Offline Compactor

Columnar layout plus gzip.

PURPOSE

Convert row-oriented sprawl into the byte layout gzip loves, ahead of time.

BOUNDARY

Offline and asynchronous. It never sits in the interactive path or blocks a user.

FAILURE PREVENTED

Slow networks dominating every session and throttled upstreams failing busy hours.

Boundary 03 / 06

Signed-URL Streaming

Bytes bypass the server heap.

PURPOSE

Move the payload from storage to browser without it ever entering server memory.

BOUNDARY

The server is an auth gate. It never sees, buffers, or transforms the payload.

FAILURE PREVENTED

Heap-limit exceptions that no retry logic can survive.

Boundary 04 / 06

Wasm-Assisted Decoder

Expand beside the heap.

PURPOSE

Expand the compressed stream into linear memory outside the garbage collector's reach.

BOUNDARY

Bytes in, typed arrays out. Parsing policy lives downstream, not here.

FAILURE PREVENTED

Multi-second tab freezes and dead tabs during cold ingestion.

Boundary 05 / 06

In-Memory Query Engine

AST filters over typed arrays.

PURPOSE

Answer nested boolean filters in milliseconds without leaving the tab.

BOUNDARY

It answers from what is loaded. It fetches nothing and persists nothing.

FAILURE PREVENTED

Input lag and server dependency for every keystroke of refinement.

Boundary 06 / 06

Delta Merge & Grid

Edits without a database.

PURPOSE

Persist user overrides as tiny versioned files merged at render time.

BOUNDARY

Single-editor snapshots. Concurrent editing gets explicit merge handling, not silent miracles.

FAILURE PREVENTED

Database-tier cost and operations for what is fundamentally a file-merge problem.

04 · FOLLOW THE FLOW

One snapshot, six hops

Watch bytes become rows become answers, with edits merging at the end.

Now let's follow one snapshot from frozen bytes to editable grid.

01 Catalog Source

The raw fixture catalog sits at over a hundred megabytes of row-oriented JSON - the honest, ugly starting point.

02 Offline Compactor

Offline, the compactor reshapes it to columnar form and gzip-compresses it to a few percent of its size, staged immutably.

03 Signed-URL Streaming

The server mints a short-lived signed URL and steps aside. The browser streams bytes directly - the CRM heap never learns they existed.

04 Wasm-Assisted Decoder

The Wasm decoder expands the stream into linear memory beside the GC heap, landing typed arrays ready for indexing. Restricted environments take the Pako path instead.

05 In-Memory Query Engine

Nested AND/OR/NOT filters compile to AST and evaluate over the arrays in milliseconds per keystroke. No server is consulted.

06 Delta Merge & Grid

User overrides serialize to kilobyte deltas and merge over base rows before paint. The grid virtualizes to a few dozen DOM rows regardless of scale.

05 · ARCHITECTURE DECISIONS

The choices that hold it up

The recorded decisions, each with the alternative it beat - then your turn.

You have seen the pipeline. Here are the recorded decisions behind it.

Wasm-assisted decode vs. main-thread JSON parsing

Expanding a multi-megabyte gzip into hundreds of thousands of JS objects on the main thread means seconds of GC pauses.

CHOSEN

Small Rust/Wasm decoder into linear memory, with a Pako fallback on the same interface.

WHY

Decoding outside the GC heap kept the UI alive during ingest; the fallback keeps the same code path working where Wasm is restricted.

ALTERNATIVES CONSIDERED

- Server-side chunking through Apex
- Main-thread loop with setTimeout slices
- Web Worker running plain JSON.parse()

Direct signed-URL streaming vs. relaying through the server

Anything that touches the CRM heap inherits its limits.

CHOSEN

Short-lived signed URL; the component fetches bytes straight from object storage.

WHY

The payload never enters server memory. The server stays an auth gate, which is all it should be here.

ALTERNATIVES CONSIDERED

- Buffering chunks through server objects
- Proxying the stream through middleware
- Moving quoting to an external portal

Delta files vs. a provisioned database for edits

Overrides need durability, but not necessarily a running database.

CHOSEN

Immutable base snapshot plus small versioned delta files merged in memory.

WHY

For single-editor quoting sessions this removes the entire database tier. Concurrent editing needs explicit merge handling - stated openly below.

ALTERNATIVES CONSIDERED

- Hosted Postgres cluster with sync API
- Document store with provisioned throughput
- Storing edits as CRM rows

Typed-array AST engine vs. `Array.filter()` per keystroke

Naive per-keystroke scans across 500K rows lag Noticeably - a few hundred milliseconds per keypress.

CHOSEN

Small lexer/parser compiling filters to AST evaluated over typed arrays and inverted indexes.

WHY

The same engine family as my json-query-lite work. Millisecond queries without leaving the tab.

ALTERNATIVES CONSIDERED

- Plain `Array.filter()` with string matching
- IndexedDB queries
- Embedded SQLite in Wasm

CHALLENGE THE DESIGN

A multi-megabyte gzip must become half a million rows. Where does that work happen?

CHUNK IT THROUGH THE SERVER

Every chunk inherits the heap ceiling that started this problem. Smaller pieces of an impossible thing are still impossible.

PARSE IT ON THE MAIN THREAD IN SLICES

Works eventually, with seconds of GC pauses and a tab users learn to fear. Slicing schedules the pain; it does not remove it.

DECODE INTO WASM LINEAR MEMORY

This is the architecture's answer. Expansion happens beside the collector, the UI stays alive, and a JS fallback covers restricted environments.

HOW THE ARCHITECT REASONED

Measure the heap you actually have before designing the pipeline - it decides everything downstream, including where decoding may live.

Users need to override prices and have it persist. What stores the overrides?

PROVISION A DATABASE

Correct for concurrent multi-user truth, and a whole tier of cost and operations for single-editor snapshots that never needed it.

STORE OVERRIDES AS CRM ROWS

Back through the per-row storage pricing that started the problem - and every override now consumes the scarcest resource in the system.

VERSIONED DELTA FILES MERGED AT RENDER

This is the architecture's answer for the single-editor case. Kilobyte files, millisecond merges, zero processes - with concurrency limits stated, not hidden.

HOW THE ARCHITECT REASONED

Delta files are a legitimate database replacement for snapshots and an honest limitation beyond them. Saying which case you serve is the architecture.

06 · FAILURE MODES

Let's try to break it

Concurrent edits, missing Wasm, and slow networks - each with a landing place.

Now let's try to break it. Each failure below lands somewhere specific - that is the design working.

The two editors at once

WHAT GOES WRONG

Two sessions override the same quote concurrently - the case deltas handle worst.

WHERE IT LANDS

Version-stamped deltas with hash checks surface the conflict for manual re-merge instead of silently winning. Optimistic concurrency, stated upfront.

The locked-down browser

WHAT GOES WRONG

A restricted environment refuses WebAssembly instantiation entirely.

WHERE IT LANDS

The identical Pako fallback decodes through the same interface. Slower, but the same code path and the same result.

The changed catalog

WHAT GOES WRONG

Upstream prices move after the snapshot compacted - the tab shows yesterday.

WHERE IT LANDS

Snapshots are versioned and freshness is a schedule, so staleness is visible metadata rather than a silent lie. Recompression is the fix, not a code change.

The thin pipe

WHAT GOES WRONG

Broadband assumptions meet a slow connection - cold load stretches.

WHERE IT LANDS

Columnar gzip keeps the transfer to a few percent of raw size, and the bottleneck is honestly the download, measurable per session.

The thin client

WHAT GOES WRONG

A low-memory device cannot comfortably hold the full working set.

WHERE IT LANDS

Typed arrays and dictionary-encoded strings minimize per-row cost, and parse buffers release immediately after indexing. Below that floor, the honest answer is a smaller snapshot scope.

STANDING COMPROMISES

Browser memory vs. server spend

Gain: No database or compute tier to operate for browsing and editing snapshots.

Cost: The tab holds the working set - on the order of 150–180MB for the largest fixture I tested.

Mitigation: Columnar typed arrays (strings as dictionary IDs, money as integer minor units) plus prompt release of parse buffers.

Wasm capability vs. sandbox reality

Gain: Near-native decode speed outside the GC heap.

Cost: Sandboxed environments restrict instantiation modes and shared memory.

Mitigation: Conservative Wasm feature set and a zero-config JS fallback; the slow path still works.

Delta simplicity vs. concurrent-write safety

Gain: No lock management, no connection pooling, full history for free.

Cost: Concurrent edits to one quote need optimistic concurrency, not row locks.

Mitigation: Version-stamped deltas with hash checks and last-writer-wins plus manual re-merge surfaced in the UI - documented as a limit, not hidden.

07 · REFLECTION

What remains true

Limits, stated plainly - and what changes next.

You have seen the pipeline, the decisions, and the failure boundaries. Two honest notes before you leave.

This pipeline is strongest for read-heavy, single-editor snapshot work: catalogs, quoting fixtures, reference data. Compression plus direct streaming plus in-memory indexing is a combination that keeps paying.

Its limits are stated, not hidden. Concurrency, freshness windows, and client RAM are real constraints with real numbers attached. The pattern transfers; the constants must be re-measured per deployment.

WHAT I WOULD CHANGE TODAY

- Progressive snapshot streaming so first paint precedes full ingestion
- Adaptive fixture scoping that sizes snapshots to measured client memory
- A stated concurrency story beyond optimistic merge for shared quotes

VERIFIED OUTCOMES

- 500K-row synthetic fixture cold-loads, decodes, and indexes in the high tens of seconds on broadband in my tests - dominated by download, not parsing
- Snapshot bytes shrink to roughly 2–3% of the row-oriented JSON on test fixtures via columnar layout plus gzip
- Interactive filtering over the full set returns in milliseconds per keystroke in the harness - no server involved
- Edits persist as KB-scale delta files with no database process to run or pay for
- Scrolling cost stays flat: constant DOM size regardless of logical row count
- Published lineage as open source: json-query-lite (TS) and a Rust/Wasm search sibling, both MIT

More in the Deep Dive Series

[01 · Generic Entity Scoring Engine, Native to the CRM](#)

[02 · BrewCMS: Open-Source Content Control Plane & Agent Operating System](#)

[03 · Local Evaluation of Salesforce RCA - CML](#)

[04 · TM Forum Open API Intelligence Suite & Spectra AI Copilot](#)

[05 · WebAssembly in Salesforce LWC: 500K Record Engine & Net-Zero DB](#)

[06 · Multi-Site Quoting Without Leaving the CRM](#)

Continue the exploration

[Interactive exhibit: victorkuldeep.com/case-studies/wasm-salesforce-lwc-500k-engine/deep-dive](https://victorkuldeep.com/case-studies/wasm-salesforce-lwc-500k-engine/deep-dive)

[Long-form essays: victorkuldeep.com/thinking](https://victorkuldeep.com/thinking)

[Ask AXIOM: victorkuldeep.com/ask](https://victorkuldeep.com/ask)

[Connect: victorkuldeep.com/connect](https://victorkuldeep.com/connect)



Kuldeep Singh

Principal Technical Architect · Salesforce Industries · Revenue Cloud (RCA) · Agentic AI

