



CASE STUDY · DEEP DIVE EXHIBIT · ~5 MIN

Multi-Site Quoting Without Leaving the CRM

“Compress the payload. Move computation to the data. Hydrate only what the eye can see.”

ROLE

Independent Architect & Builder

TIMELINE

Independent research · 2025–2026

DOMAIN

High-Volume Platform Architecture

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Independent research on synthetic quoting fixtures - no client data, no employer systems. Where the catalog-engine study covers storage and decoding, this one covers the last mile: keeping a CRM-embedded quoting UI native and responsive while the dataset behind it is large. Pako gzip via static resources for transport, a Rust indexer and a TypeScript in-memory engine (both from my json-query-lite lineage) for retrieval, and incremental hydration so the component renders dozens of rows instead of tens of thousands. Same independence note as throughout: built on my own time, generic fixtures, nothing proprietary.

Independent work on generated fixtures. Multi-site quoting is a good stress case because it combines three pains at once: many locations, many attributes per location, and users who refuse to leave the CRM to do the work. I built a synthetic version of that problem to test one claim - that a native component can stay native if it never holds the whole dataset.

What breaks when the dataset outgrows the component: transferring full payloads freezes the UI thread, holding everything in memory exhausts the tab, and paging through a server per keystroke feels sluggish. The temptation is an external portal - which users experience as abandonment.

CONSTRAINTS

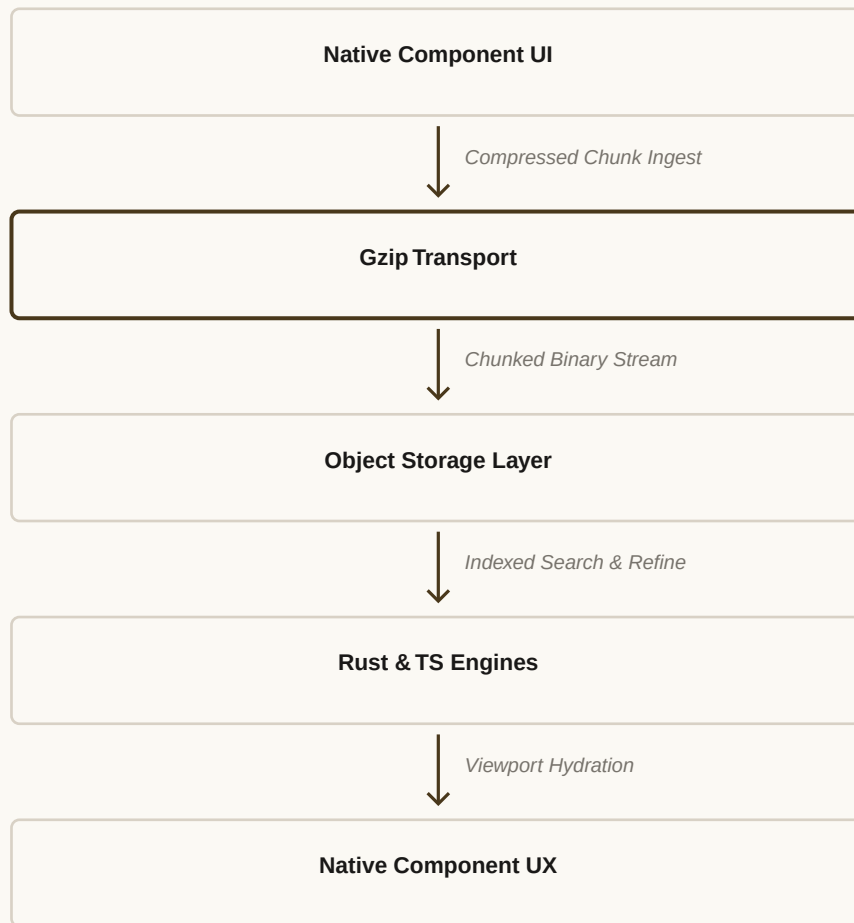
- Quoting stays inside the CRM component - no portal redirect
- No tab crashes or multi-second freezes on large fixtures
- Respect sandbox transport limits and security boundaries
- Search across locations and attributes must feel instant

PART I · ARCHITECTURE

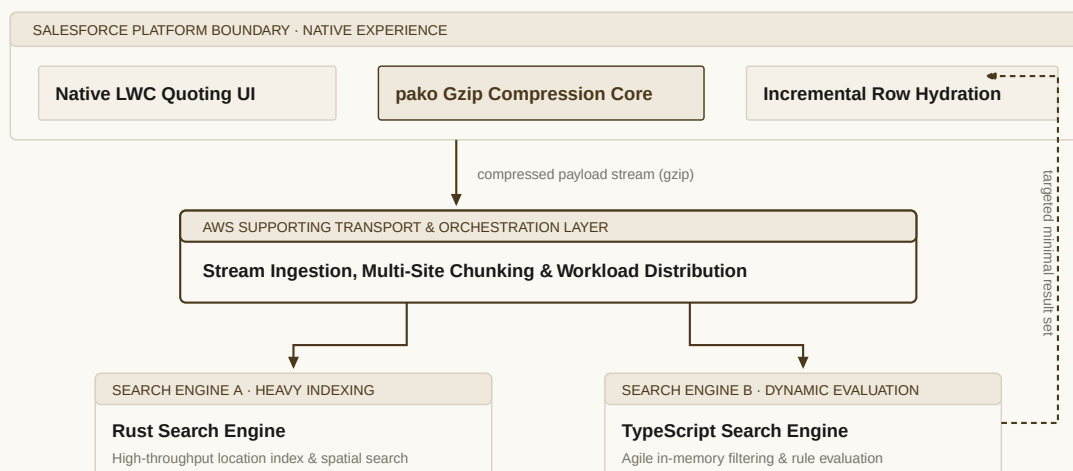
The system as designed

Compress on the way in, index outside the render path, hydrate incrementally. The component loads Pako from static resources and works in array buffers, not JSON strings. An async transport layer pages compressed chunks; the Rust indexer handles heavy scans while the TypeScript engine answers interactive refinements in memory. Only the visible slice - ranked IDs plus display fields - reaches the DOM.

HIGH-VOLUME QUOTING & TARGETED HYDRATION



Compressed chunks travel once; engines answer locally; the component renders only the visible slice.



Full system schematic - High-Volume Quoting & Targeted Hydration.

Choices, engineering, tradeoffs

Client-side gzip vs. moving quoting off-platform

The industry default for oversized payloads is an external app. Users hate the context switch.

CHOSEN

In-browser gzip via static resources with a compressed transport layer.

WHY

Keeps reps in the workflow they know while cutting wire bytes dramatically on repetitive catalog data.

Two engines with distinct jobs vs. one store doing everything

Bulk indexing and interactive refinement stress different resources.

CHOSEN

Rust indexer for heavy scans plus TypeScript engine for in-memory refinement.

WHY

Each does what it is good at. A small router sends bulk scans one way and keystroke refinements the other, over a shared contract.

ENGINEERING NOTES

- Loaded Pako as a static resource and kept payloads as array buffers end to end - no JSON string round-trips.
- Chunked ingest with cooperative yielding so decompression never blocks input handling.
- Built the transport as a thin chunking and orchestration layer over object storage - no database, no session affinity.
- Used the Rust indexer for high-cardinality scans and json-query-lite in memory for refinements and ranking.
- Rendered incrementally: the component holds result IDs and hydrates display rows for the viewport only.
- Kept a shared record contract between engines so either can answer a query class without the UI caring.

PART I · OUTCOMES

Tradeoffs, results, lessons

Decompression CPU vs. transfer time

Gain: Far fewer bytes cross the network; payload caps stop biting.

Cost: The client spends cycles expanding gzip.

Mitigation: Chunked, yielded decompression scheduled around input frames.

Two-language stack vs. one

Gain: Bulk throughput plus UI agility.

Cost: Two toolchains to govern.

Mitigation: One record contract and one query grammar shared by both engines.

OUTCOMES

- Quoting stays in the native component across tested fixtures - no portal redirect in the flow
- Wire bytes fall sharply on repetitive catalog fixtures versus uncompressed JSON
- Interactive refinements answer from memory without a server round-trip
- Render cost stays flat via viewport hydration regardless of result-set size

LESSONS

- Decide where bytes rest, where they move, and where they compute - in that order.
- Static resources plus disciplined buffers take a CRM component surprisingly far.

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 keep a quoting session inside the CRM while the dataset behind it grows enormous.

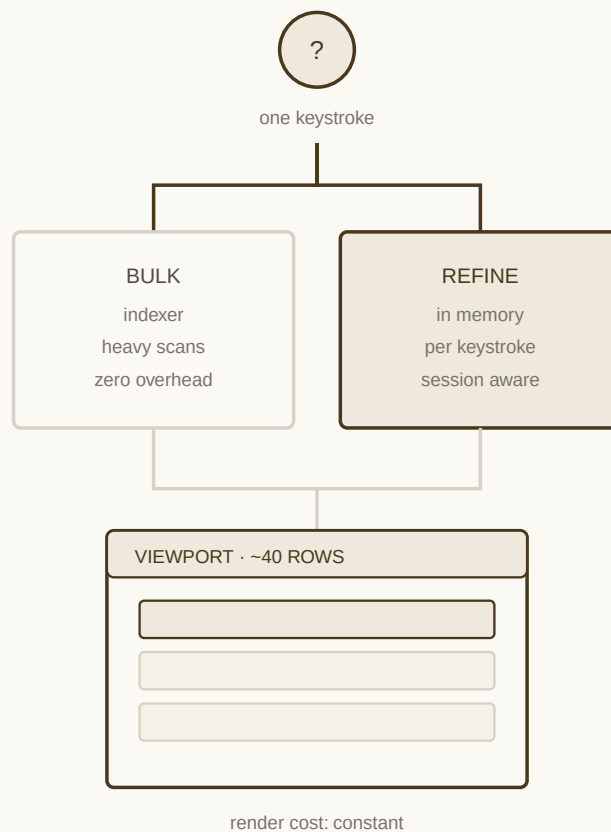


Exhibit plate - the system at a glance.

This exhibit is the companion to the catalog-engine study: where that one covers storage and decoding, this one covers the last mile. Compressed chunks travel once, two search engines divide the work by temperament, and the component renders dozens of rows instead of tens of thousands.

Built on generated fixtures with no client data and no employer systems. The claim under test is narrow: a native component can stay native if it never holds the whole dataset.

How to use this exhibit

- Inspect each link in the transport-to-render chain
- Step one query from keystroke to visible rows
- Decide which engine should answer which question
- Break the transport, then watch hydration degrade gracefully

02 · THE PROBLEM

The dataset versus the tab

Full payloads freeze threads, full retention exhausts memory, and portals abandon users.

Large datasets push quoting out of the CRM. Users experience that as abandonment.

Multi-site quoting combines three pains: many locations, many attributes per location, and users who refuse to leave the CRM to do the work. Full payloads freeze the UI thread, full retention exhausts the tab, and per-keystroke server paging feels sluggish.

The industry default - an external portal - solves the engineering problem by moving it onto the user. This exhibit refuses that trade.

THE CONSTRAINTS THIS FAILURE IMPOSES

- Quoting stays inside the CRM component - no portal redirect
- No tab crashes or multi-second freezes on large fixtures
- Respect sandbox transport limits and security boundaries
- Search across locations and attributes must feel instant

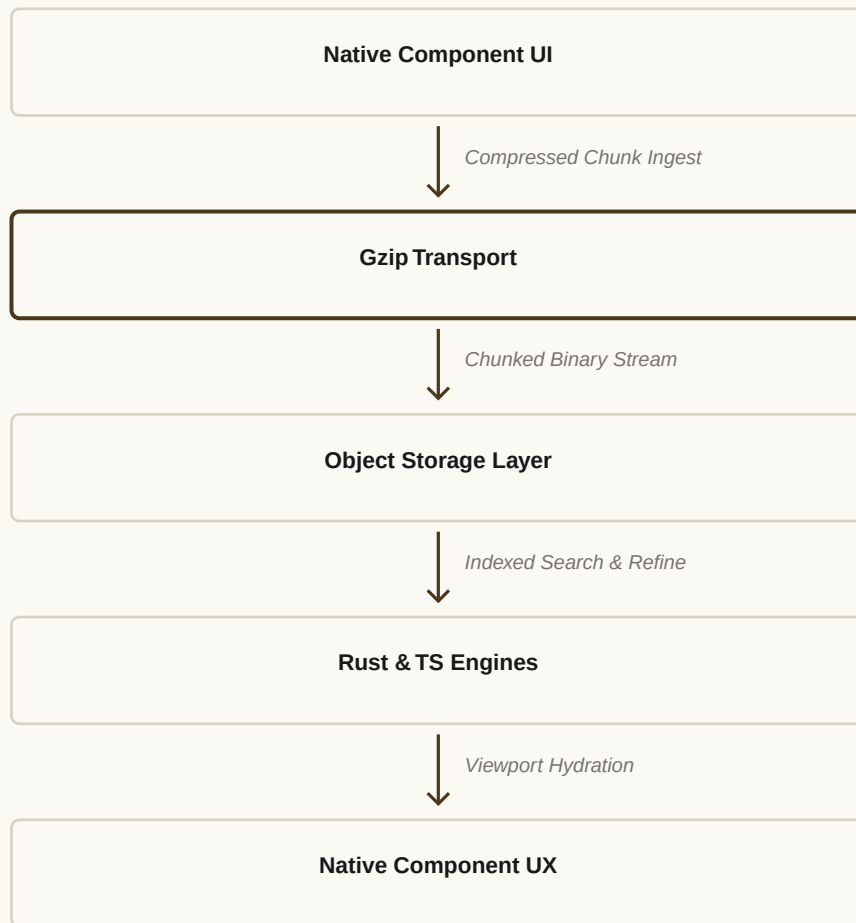
03 · THE ARCHITECTURE

Inspect the system

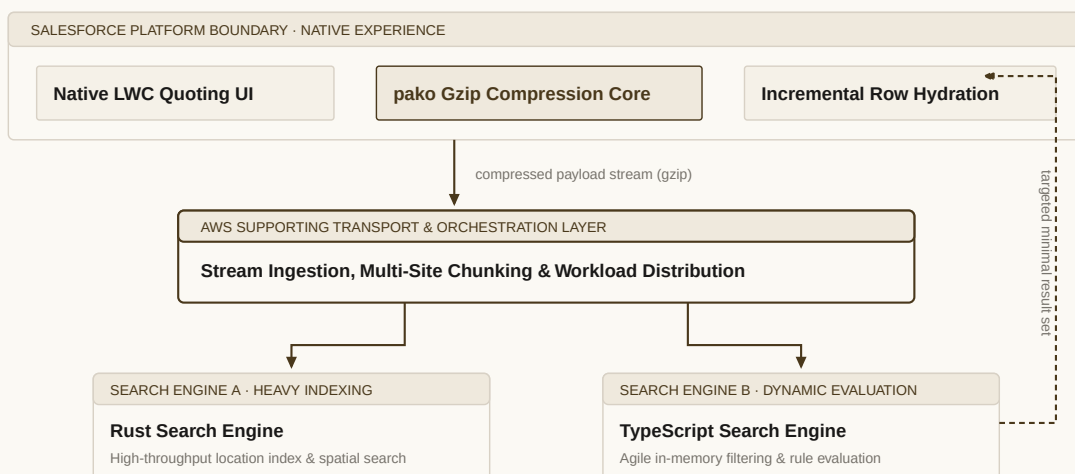
From keystroke to visible rows through compression, two engines, and a viewport.

Compress once, index by temperament, hydrate only the viewport.

HIGH-VOLUME QUOTING & TARGETED HYDRATION



Compressed chunks travel once; engines answer locally; the component renders only the visible slice.



Full system schematic - High-Volume Quoting & Targeted Hydration.

Boundary 01 / 06

Quoting Interface

Native component, full context.

PURPOSE

Keep reps inside the workflow they know, with CRM context intact.

BOUNDARY

It asks questions and shows answers. It holds no dataset and runs no indexes.

FAILURE PREVENTED

Workflow abandonment disguised as a platform migration.

Boundary 02 / 06

Gzip Transport

Pako chunks, array buffers.

PURPOSE

Shrink repetitive catalog data before it ever touches the network.

BOUNDARY

Bytes in transit only. It stores nothing and decides nothing about relevance.

FAILURE PREVENTED

Payload-cap failures and multi-second transit stalls on repetitive data.

Boundary 03 / 06

Chunked Object Storage

Compressed chunks at rest.

PURPOSE

Hold the dataset as addressable compressed chunks instead of rows in a database.

BOUNDARY

Dumb storage, deliberately. All intelligence lives in the engines and the component.

FAILURE PREVENTED

Database-tier cost, connection management, and sync machinery for static snapshots.

Boundary 04 / 06

Rust Bulk Indexer

Heavy scans, zero overhead.

PURPOSE

Absorb the scans that would crush an interpreter: high-cardinality sweeps across the corpus.

BOUNDARY

Retrieval only. Ranking nuance and session context belong to the refinement engine.

FAILURE PREVENTED

Multi-second full-corpus scans blocking interactive refinement.

Boundary 05 / 06

TypeScript Refiner

Keystroke agility in memory.

PURPOSE

Answer interactive refinements from memory while the user is still typing.

BOUNDARY

Refinement, not discovery. It re-ranks and narrows; the indexer finds.

FAILURE PREVENTED

Round-trips per keystroke and the input lag users read as brokenness.

Boundary 06 / 06

Viewport Hydration

Render dozens, not thousands.

PURPOSE

Keep rendering cost constant no matter how large the result set grows.

BOUNDARY

A sliding window, nothing more. Scroll position is state; off-screen rows are not.

FAILURE PREVENTED

Render-cost blowups that turn large result sets into frozen viewports.

04 · FOLLOW THE FLOW

One query, six hops

Watch a keystroke become a handful of rendered rows.

Now let's follow one keystroke to a rendered viewport.

01 Quoting Interface

A rep types a location filter. The component captures intent and debounces - it reaches for no dataset of its own.

02 Gzip Transport

Needed chunks arrive compressed and expand cooperatively around input frames. The wire carries a fraction of the JSON equivalent.

03 Chunked Object Storage

Chunks serve versioned and stateless. No sessions, no queries at rest - just addressable bytes.

04 Rust Bulk Indexer

The bulk scan runs over inverted indexes: the heavy first pass across the corpus, over in milliseconds.

05 TypeScript Refiner

The refinement engine narrows and ranks within the working set, session-aware, per keystroke.

06 Viewport Hydration

Ranked identifiers hydrate into display rows for the visible window only. The DOM never learns how large the answer was.

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 chain. Here are the recorded decisions behind it.

Client-side gzip vs. moving quoting off-platform

The industry default for oversized payloads is an external app. Users hate the context switch.

CHOSEN

In-browser gzip via static resources with a compressed transport layer.

WHY

Keeps reps in the workflow they know while cutting wire bytes dramatically on repetitive catalog data.

ALTERNATIVES CONSIDERED

- Standalone external quoting app
- Dozens of paged server queries per interaction
- Uncompressed batched polling

Two engines with distinct jobs vs. one store doing everything

Bulk indexing and interactive refinement stress different resources.

CHOSEN

Rust indexer for heavy scans plus TypeScript engine for in-memory refinement.

WHY

Each does what it is good at. A small router sends bulk scans one way and keystroke refinements the other, over a shared contract.

ALTERNATIVES CONSIDERED

- One monolithic query path
- Pure client-side array scans

CHALLENGE THE DESIGN

A query needs a bulk corpus scan plus interactive refinement. One engine or two?

ONE ENGINE FOR EVERYTHING

Simpler to own - and mediocre at both jobs. Bulk scans want predictability, keystrokes want agility, and one runtime compromises each.

SERVER-SIDE EVERYTHING PER KEYSTROKE

Unlimited compute per keypress, purchased with latency on every press. Users feel each round-trip as hesitation.

TWO ENGINES BY TEMPERAMENT

This is the architecture's answer. Rust absorbs bulk scans, TypeScript owns refinement, and a router sends each question to its natural home over one shared contract.

HOW THE ARCHITECT REASONED

Assign each workload to its ideal runtime over a shared record contract, and the two-language cost collapses to an interface.

Payloads exceed CRM tolerances. The standard advice is an external portal. Take it?

BUILD THE EXTERNAL PORTAL

Engineering relief purchased with user abandonment. Context, workflow, and trust stay in the CRM while the work moves elsewhere.

DOZENS OF PAGED SERVER QUERIES

Keeps users in place while making every interaction a multi-round-trip saga. Slowness distributed is still slowness.

COMPRESS TRANSPORT, HYDRATE VIEWPORTS

This is the architecture's answer. The dataset shrinks on the wire and the DOM stays constant - the component never holds what it cannot afford.

HOW THE ARCHITECT REASONED

Decide where bytes rest, where they move, and where they compute - in that order. Most portal migrations never asked the question.

06 · FAILURE MODES

Let's try to break it

Dropped chunks, cold indexes, and result floods - each with a landing place.

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

The dropped chunk

WHAT GOES WRONG

A compressed chunk fails in transit mid-session.

WHERE IT LANDS

Chunks are versioned and addressable, so the transport retries exactly the missing bytes. The viewport shows its current state, never a torn one.

The cold start

WHAT GOES WRONG

First query arrives with nothing indexed and a user already typing.

WHERE IT LANDS

The refinement engine answers from whatever is resident while bulk indexing completes - degraded breadth first, full depth shortly after.

The million-row answer

WHAT GOES WRONG

A filter matches nearly everything - the nightmare case for rendering.

WHERE IT LANDS

Viewport hydration renders the visible window regardless. Result-set size is a number in the header, not DOM nodes in the page.

The changed catalog

WHAT GOES WRONG

Source data moves while chunks sit cached in sessions.

WHERE IT LANDS

Versioned chunks make staleness explicit metadata. Refresh targets versions, not vibes.

The weak device

WHAT GOES WRONG

Decompression and refinement on hardware with little to spare.

WHERE IT LANDS

Yielded chunked decompression keeps input responsive, and the working set is contract-bounded. Below the floor, scope narrows explicitly.

STANDING COMPROMISES

Decompression CPU vs. transfer time

Gain: Far fewer bytes cross the network; payload caps stop biting.

Cost: The client spends cycles expanding gzip.

Mitigation: Chunked, yielded decompression scheduled around input frames.

Two-language stack vs. one

Gain: Bulk throughput plus UI agility.

Cost: Two toolchains to govern.

Mitigation: One record contract and one query grammar shared by both engines.

07 · REFLECTION

What remains true

Limits, stated plainly - and what changes next.

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

This chain is strongest where datasets are large, repetitive, and read-mostly: catalogs, quoting fixtures, reference corpora. Compression plus temperament-split engines plus viewport rendering is a combination that keeps paying.

Its limits are stated, not hidden. Write-heavy collaboration, sub-second freshness demands, and truly tiny devices each break an assumption here - and each breakage is named above rather than discovered later.

WHAT I WOULD CHANGE TODAY

- Prefetch chunks by predicted filter paths instead of on-demand fetch

- Unify the two engines behind one query planner with cost-based routing
- Add explicit offline scope so disconnected sessions degrade by design

VERIFIED OUTCOMES

- Quoting stays in the native component across tested fixtures - no portal redirect in the flow
- Wire bytes fall sharply on repetitive catalog fixtures versus uncompressed JSON
- Interactive refinements answer from memory without a server round-trip
- Render cost stays flat via viewport hydration regardless of result-set size

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/connectbase-high-volume-quoting/deep-dive

Long-form essays: victorkuldeep.com/thinking

Ask AXIOM: victorkuldeep.com/ask

Connect: victorkuldeep.com/connect



Kuldeep Singh

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

