



SYSTEMS ARCHITECTURE & PRODUCTION ENGINEERING
· COMPLETE VOLUME

Technical Case Studies & Deep Dive Series

“A comprehensive compendium of 12 production architectures, reactive compilers, CRM runtime engines, agent control planes, and distributed systems.”

AUTHOR	ROLE	MONOGRAPHS	EDITION
Kuldeep Singh	Principal Systems Architect	12 Case Studies (6 Deep Dives)	2026 Archive Edition

KULDEEP SINGH · DIGITAL ARCHITECTURE · COMPLETE SERIES

Architectural Monographs & Deep Dive Index

Index of 12 verified technical monographs documenting architecture decisions, constraints, and runtime designs.

01	Headless Salesforce Revenue Cloud (RCA) Capabilities, Without Dual Maintenance	CASE STUDY
	Headless Platform Architecture · Independent research · 2025–2026	
02	Local Evaluation of Salesforce RCA - CML	CASE STUDY + DEEP DIVE
	Compiler & Runtime Engineering · Independent research · 2025–2026	
03	Generic Entity Scoring Engine, Native to the CRM	CASE STUDY + DEEP DIVE
	Agentic Systems & AI Architecture · Independent research · 2025–2026	
04	TM Forum Open API Intelligence Suite & Spectra AI Copilot	CASE STUDY + DEEP DIVE
	Telco Systems & Applied AI Architecture · Architecture & Open Systems Initiative · 2026	
05	BrewCMS: Open-Source Content Control Plane & Agent Operating System	CASE STUDY + DEEP DIVE
	Open Source & Content Systems · Open Source Systems Initiative · 2026	
06	WebAssembly in Salesforce LWC: 500K Record Engine & Net-Zero DB	CASE STUDY + DEEP DIVE
	Systems Engineering & High-Volume Web Architecture · Independent research · 2025–2026	
07	Multi-Site Quoting Without Leaving the CRM	CASE STUDY + DEEP DIVE
	High-Volume Platform Architecture · Independent research · 2025–2026	
08	Save Pipelines Drawn as Graphs, Not Trigger Chains	CASE STUDY
	Salesforce Revenue Cloud (RCA) & Workflow Architecture · Independent research · 2025–2026	
09	A CRM-Backed Portal That Reads From the Edge	CASE STUDY
	Platform Architecture · Independent research · 2025–2026	
10	Tokenized Enterprise Design System & Micro-Frontend Architecture	CASE STUDY
	Experience Architecture · Architecture Initiative	
11	Resilient Event-Driven Integration & Distributed Outbox Topology	CASE STUDY
	Integration Architecture · Platform Transformation	
12	json-query-lite: SQL-Shaped Search for Big JSON in Small Places	CASE STUDY
	Search & Retrieval Systems · Open source · independently maintained	



Headless Salesforce Revenue Cloud (RCA) Capabilities, Without Dual Maintenance

“Separate capability from presentation - but keep exactly one engine that owns the truth.”

ROLE	TIMELINE	DOMAIN
Independent Architect	Independent research · 2025–2026	Headless Platform Architecture

PART I · THE CASE

What was built, and why

Independent pattern study with demo contracts I authored - no employer APIs, no client schemas. The setup under test: a Salesforce Revenue Cloud (RCA) engine (pricing, configuration, rule evaluation) stays authoritative on the platform side, while React storefronts, partner surfaces, and internal tools consume it through versioned, contract-first endpoints. A gateway mediates auth, limits, and aggregation; edge caching absorbs catalog reads while transactional calculations pass through. The point of the study is the failure it avoids: rewriting pricing in a Node service and discovering the drift in production.

Independent work, demo contracts only. Revenue logic has to reach many surfaces - self-service, partner, mobile, internal - and every surface team eventually proposes owning a copy. I modeled the alternative I trust: one engine, many readers, contracts in between.

Binding engine logic to internal page layouts leaves two bad options: force outsiders onto internal screens, or copy the logic outward and maintain both. Copies always diverge - usually discovered as misquotes.

CONSTRAINTS

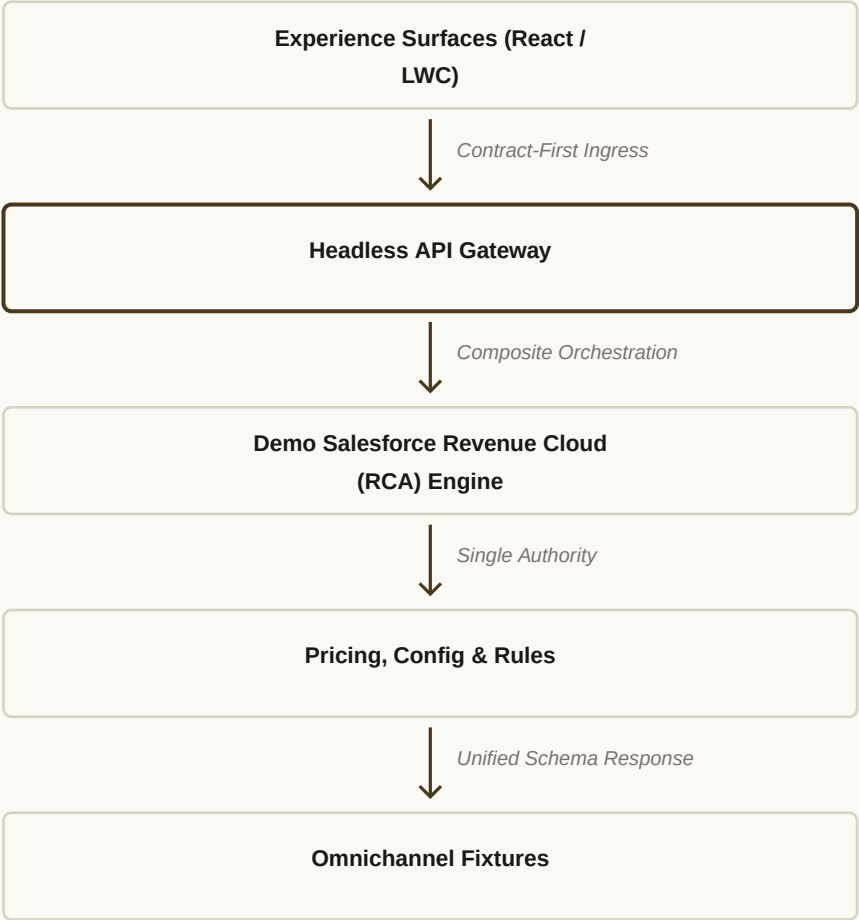
- One authoritative engine for price, configuration, and contract rules in the study model
- External consumers integrate only through versioned contracts
- Public-traffic patterns must respect platform rate limits
- Catalog reads stay fast; transactional calculations stay exact

PART I · ARCHITECTURE

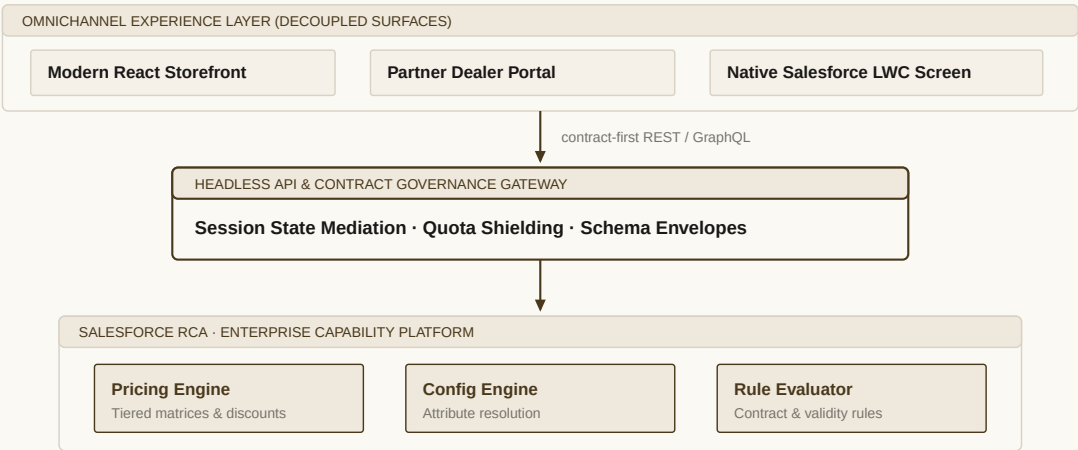
The system as designed

A gateway sits between experience surfaces and engine capabilities. It translates auth, enforces budgets, and aggregates multi-step inquiries into single-turn composite calls. Catalog metadata rides edge cache with contract-aware invalidation; live calculations pass through to the engine. Client libraries for React and web components share the same schemas.

HEADLESS SALESFORCE REVENUE CLOUD (RCA) CAPABILITY STUDY



One demo engine, versioned contracts, edge-cached catalog with live transactional calculation.



Full system schematic - Headless Salesforce Revenue Cloud (RCA) Capability Study.

Choices, engineering, tradeoffs

Headless capability layer vs. rewriting the engine outside

Rewriting pricing externally is fast to start and slow to own.

CHOSEN

Keep the study-model engine authoritative; expose it through contracts.

WHY

One implementation to fix, many surfaces to serve. Frontend teams keep full visual freedom without owning arithmetic.

Edge caching for catalog, pass-through for transactions

Reads and calculations have different freshness needs.

CHOSEN

Cache catalog dimensions at the edge; calculate transactions live.

WHY

Browsing stays instant and cheap while money math stays exact.

ENGINEERING NOTES

- Authored demo REST/GraphQL contracts that hide internal object shapes behind stable fields.
- Built the gateway concerns explicitly: auth translation, per-route budgets, and response aggregation.
- Designed composite endpoints so one configurator turn costs one call, not five.
- Shipped thin client libraries sharing the same schemas across React and web-component consumers.

PART I · OUTCOMES

Tradeoffs, results, lessons

Gateway ownership vs. direct calls

Gain: Uniform contracts, budgets, and auth across channels.

Cost: A gateway to run and version.

Mitigation: Contract tests in CI that fail the build on breaking schema drift.

Live calculation vs. limit budgets

Gain: Exact quotes on every transaction.

Cost: Public traffic can press platform allowances.

Mitigation: Cache everything cacheable; reserve live calls for money math with session-scoped tokens.

OUTCOMES

- Demo engine serves web, partner, and internal fixture clients from one contract set
- No duplicated pricing logic across demo surfaces
- Catalog browsing served from edge in the study setup; transactions calculate live
- Breaking contract changes surface in CI before deployment in the harness

LESSONS

- Keep one engine authoritative; let as many surfaces read it as contracts allow.
- Cache what is stable, calculate what involves money - and write down which is which.

KULDEEP SINGH



CASE STUDY 02 · DEEP DIVE EXHIBIT · ~6 MIN

Local Evaluation of Salesforce RCA - CML

“When every configurator click pays a server round-trip, move evaluation to the client - but do it with a real language runtime, not duplicated if-statements.”

ROLE

Independent Architect & Builder

TIMELINE

Independent research · 2025–2026

DOMAIN

Compiler & Runtime Engineering

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Independent weekend work, built on my own time and equipment. The recurring pattern: Salesforce Revenue Cloud (RCA) configurators evaluate constraint rules on the server, so each option toggle or attribute change costs a remote call - often hundreds of milliseconds, with out-of-order responses stepping on newer choices. I built a small TypeScript compiler pipeline - lexer, recursive-descent parser, typed AST, deterministic tree-walking evaluator - that runs the same rule set in browser memory. One codebase ships as ESM for React storefronts and as an IIFE/static-resource bundle for LWC. Interactive evaluation drops to single-digit milliseconds in my harness, and there is exactly one rule implementation to maintain. No employer or client material is involved; the design is deliberately generic and the verification harness is my own.

This is independent work - no employer code, no client config, no proprietary schemas. I kept seeing the same shape in enterprise quoting: a product model with hundreds of interdependent rules (eligibility, co-requisites, pricing attributes), one engine on the server, and two or three hand-maintained copies of the logic leaking into storefront JavaScript. I wanted to know whether a single, honest compiler core could serve both a CRM-embedded UI and a headless React surface without drift. So I built it as a weekend project, against a rule grammar I defined myself, informed only by public documentation patterns.

The failure mode I kept observing in Salesforce Revenue Cloud (RCA) configurators:

- 1) Per-click round-trips: Every toggle or attribute edit goes remote. At 300–800ms a click, fast configurators feel broken, and overlapping responses overwrite newer state.
- 2) Headless gets second-class validation: The storefront either polls the server per keystroke (slow, rate-limit hungry) or re-implements rules by hand.
- 3) Hand ports drift: Two implementations of the same pricing logic always diverge. The bug reports arrive months later, as misquotes.
- 4) Sandbox says no to shortcuts: In LWC, `eval()` and `new Function()` are off the table under Lightning Web Security, so 'just compile to JS strings' is not an option.

CONSTRAINTS

- No server calls during interaction: once rules are loaded, option and attribute changes evaluate locally
- One rule implementation: the LWC path and the React path execute the same AST, not two ports
- Sandbox-safe: no `eval()`, no `new Function()`, strict CSP-compatible
- Fast enough to disappear: multi-rule evaluation in low single-digit milliseconds on commodity hardware
- Small and shippable: zero-dependency bundle (~30KB class) via static resources and NPM

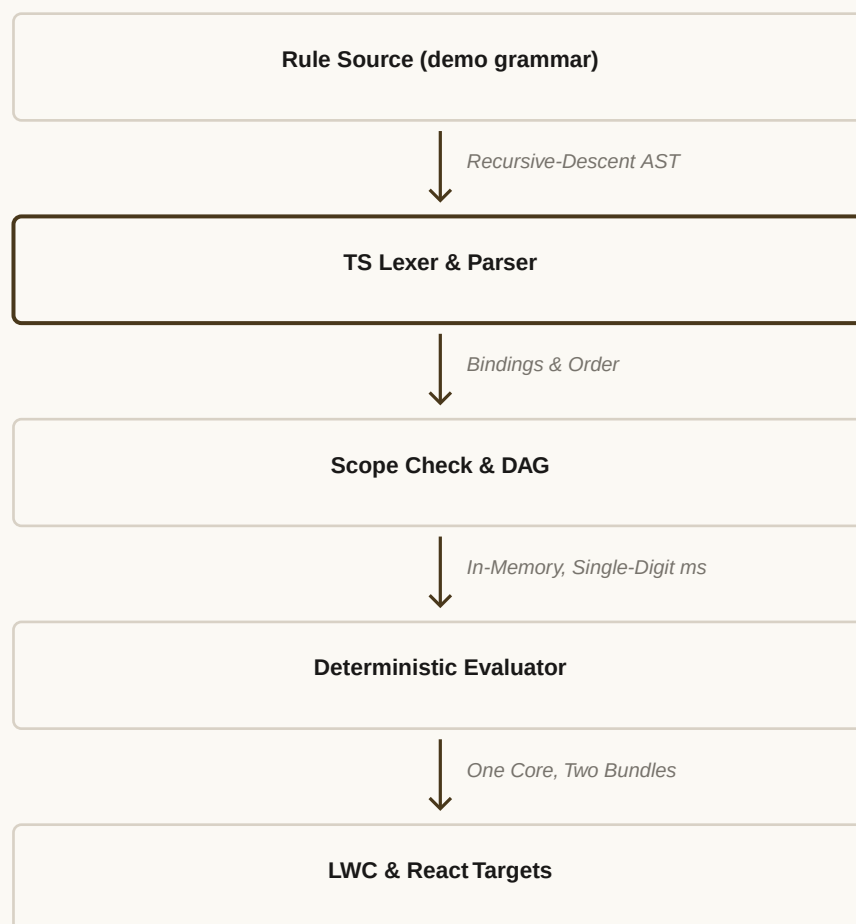
PART I · ARCHITECTURE

The system as designed

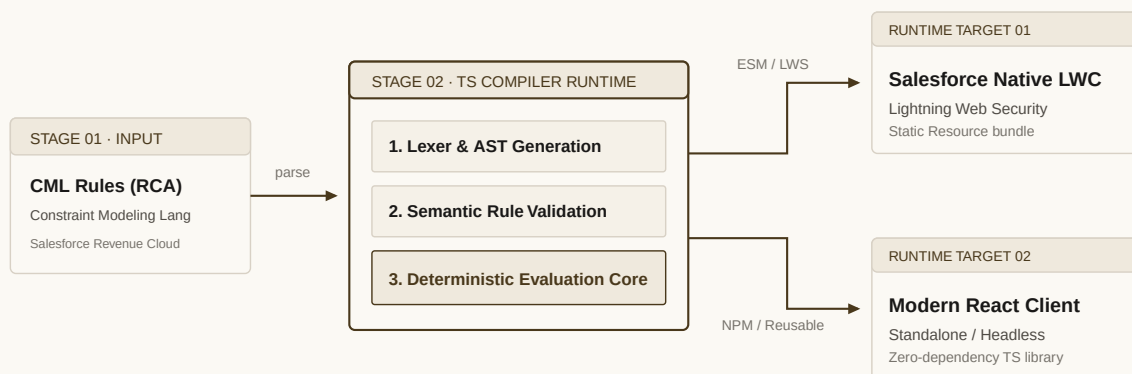
Four stages, all in plain TypeScript, no dependencies:

1. Scanner & parser: a hand-written lexer and recursive-descent parser turns rule text into a typed AST - precedence, negation, nested booleans, function calls. Nothing clever, just careful.
2. Scope check & dependency graph: variable bindings resolve against a catalog schema I define per demo model, producing a DAG so cascading attributes (A changes B changes C) evaluate in topological order with cycle detection.
3. Tree-walking evaluator: a small interpreter with deterministic coercion and memoised subtrees. Boring on purpose - the same input always produces the same output, on either runtime.
4. Two packages, one core: ESM for React/Next.js, IIFE/UMD for LWC via platformResourceLoader. A differential harness runs thousands of generated cases through both and diffs the results.

CONSTRAINT COMPILER & DETERMINISTIC EVALUATION



Declarative rules compile to a typed AST once, then evaluate in memory on either runtime - no per-click server calls.



Full system schematic - Constraint Compiler & Deterministic Evaluation.

PART I · DECISIONS

Choices, engineering, tradeoffs

Client-side evaluation core vs. per-click server calls

The default I saw everywhere was remote evaluation per interaction - correct, but slow and racy.

CHOSEN

A real compiler core in dependency-free TypeScript, evaluated in memory.

WHY

The network was the budget, not the CPU. Moving evaluation client-side removed the round-trip entirely; keeping it a proper AST (not string hacks) kept semantics identical across runtimes.

Tree-walking interpreter vs. generating JS functions

`new Function()` would be faster on paper, but LWS and enterprise CSP forbid it - and rightly so.

CHOSEN

AST interpreter with deterministic coercion and node memoisation.

WHY

Compliance first, then speed. Short-circuit booleans and memoised subtrees kept my test models in the low single-digit milliseconds, which is fast enough that nobody notices the interpreter.

One shared library vs. Apex-plus-JavaScript dual maintenance

Every team I watched tried the dual implementation at least once. Every one of them paid for it in misquotes.

CHOSEN

A single canonical TS library consumed by both LWC and React.

WHY

One grammar, one parser, one evaluator. Drift becomes structurally impossible rather than process-prevented.

Topological DAG resolution vs. multi-pass dirty-checking

Cascading attributes loop forever under naive re-evaluation.

CHOSEN

DAG with topological order and cycle detection at load time.

WHY

Single pass, deterministic order, and circular rules fail loudly at authoring time instead of freezing a UI thread.

ENGINEERING NOTES

- Hand-wrote the scanner and parser - no generator dependencies - covering arithmetic, comparisons, nested AND/OR/NOT, and domain functions, with error messages that point at the offending token.
- Built the dependency graph as an adjacency list over attribute references; Kahn's algorithm gives the evaluation order and rejects cycles before anything renders.
- Kept the evaluator allocation-light: pre-sized scopes, memoised pure subtrees, and no code generation of any kind, so it loads cleanly under LWS.
- Shipped two artifacts from one source: ESM for bundlers and a minified IIFE for LWC static resources; the LWC wrapper is a thin loader, not a fork.
- Wrote a differential harness that generates thousands of rule/input combinations and asserts byte-identical outcomes between the LWC bundle and the React import - that harness is the real parity guarantee.
- Added a build-time rule linter that flags unknown attributes, circular references, and dead constraints before deployment, because authoring errors are cheaper than runtime ones.

PART I · OUTCOMES

Tradeoffs, results, lessons

Bundle weight vs. network chatter

Gain: Interaction evaluation happens in memory with no network at all.

Cost: The client downloads and parses a small compiler library up front.

Mitigation: Kept it dependency-free and async-loadable; on my test models it parses in well under a frame.

Bounded grammar vs. full macro language

Gain: Evaluation time stays predictable - no runaway recursion in the UI thread.

Cost: No unbounded recursive macros in the rule language.

Mitigation: Everything I actually needed in configurators fits declarative constraints; macros were never worth the risk.

Interpreter overhead vs. JIT-compiled functions

Gain: Fully sandbox- and CSP-safe with no dynamic code.

Cost: Tree-walking is slower than native JIT per node.

Mitigation: Memoisation plus short-circuiting keeps realistic models in the low milliseconds - the network saving dwarfs it.

OUTCOMES

- Interactive evaluation moved from per-click server latency to in-memory execution (low single-digit ms across my differential harness models)
- One rule implementation serves both LWC and React paths - verified by differential tests, not by convention
- Authoring errors surface at build time (unknown attributes, cycles, dead rules) instead of as production misquotes
- Sandbox-safe by construction: zero dynamic code generation, loads under LWS and strict CSP
- Reusable across demo models without modification - the engine knows nothing about any employer's catalog

LESSONS

- When the platform boundary is the bottleneck, owning a small compiler beats adding another cache layer.
- A differential test harness across runtimes is worth more than any parity claim in prose.
- Interpreters earn their keep in sandboxed environments: predictable, auditable, and fast enough.

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 watch a language come alive inside the browser - compiled, not called.

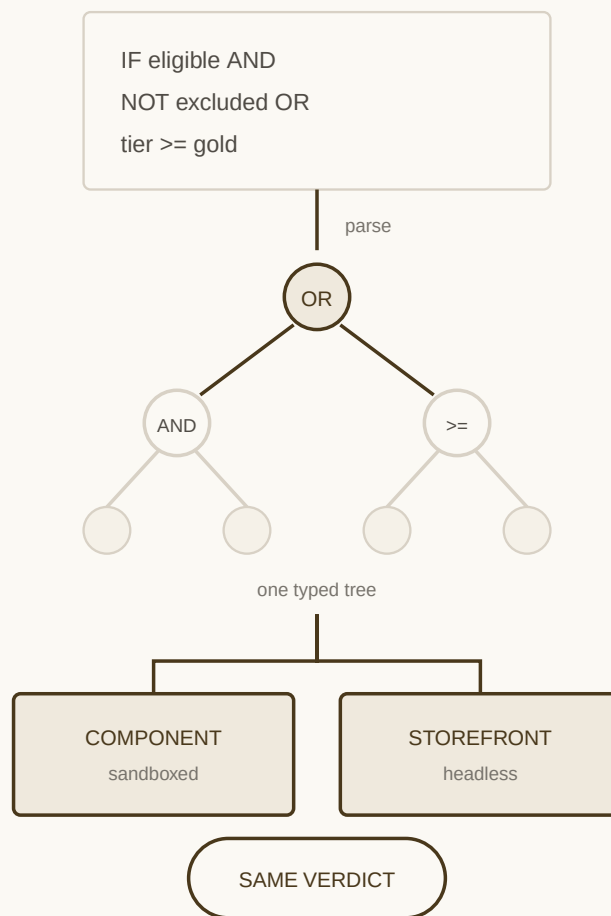


Exhibit plate - the system at a glance.

This exhibit traces a small constraint-language compiler: declarative product rules enter as text, a hand-written pipeline turns them into a typed tree, and a deterministic evaluator runs them in browser memory on two runtimes that must never disagree.

It is independent weekend work against a self-defined grammar, informed only by public documentation patterns. The verification harness is the real artifact - thousands of generated cases asserting identical outcomes on both targets.

How to use this exhibit

- Inspect each compiler stage to see what it guarantees
- Step one rule from source text to evaluated result
- Decide between generating code and interpreting trees
- Break the grammar, then watch where each break lands

02 · THE PROBLEM

The round-trip tax

Hundreds of milliseconds per click, drifting hand ports, and a sandbox that bans shortcuts.

Every click pays a server round-trip. That is not a UI problem - it is an evaluation-placement problem.

Salesforce Revenue Cloud (RCA) configurators evaluate constraint rules on the server, so each toggle or attribute change costs a remote call - hundreds of milliseconds, with overlapping responses overwriting newer choices. Storefronts either poll per keystroke or hand-maintain a second copy of the rules.

Hand ports always drift. The misquotes arrive months later. And the fast shortcut - generating JavaScript from rules with eval - is forbidden inside sandboxed component runtimes.

THE CONSTRAINTS THIS FAILURE IMPOSES

- No server calls during interaction: once rules are loaded, option and attribute changes evaluate locally
- One rule implementation: the LWC path and the React path execute the same AST, not two ports
- Sandbox-safe: no eval(), no new Function(), strict CSP-compatible
- Fast enough to disappear: multi-rule evaluation in low single-digit milliseconds on commodity hardware
- Small and shippable: zero-dependency bundle (~30KB class) via static resources and NPM

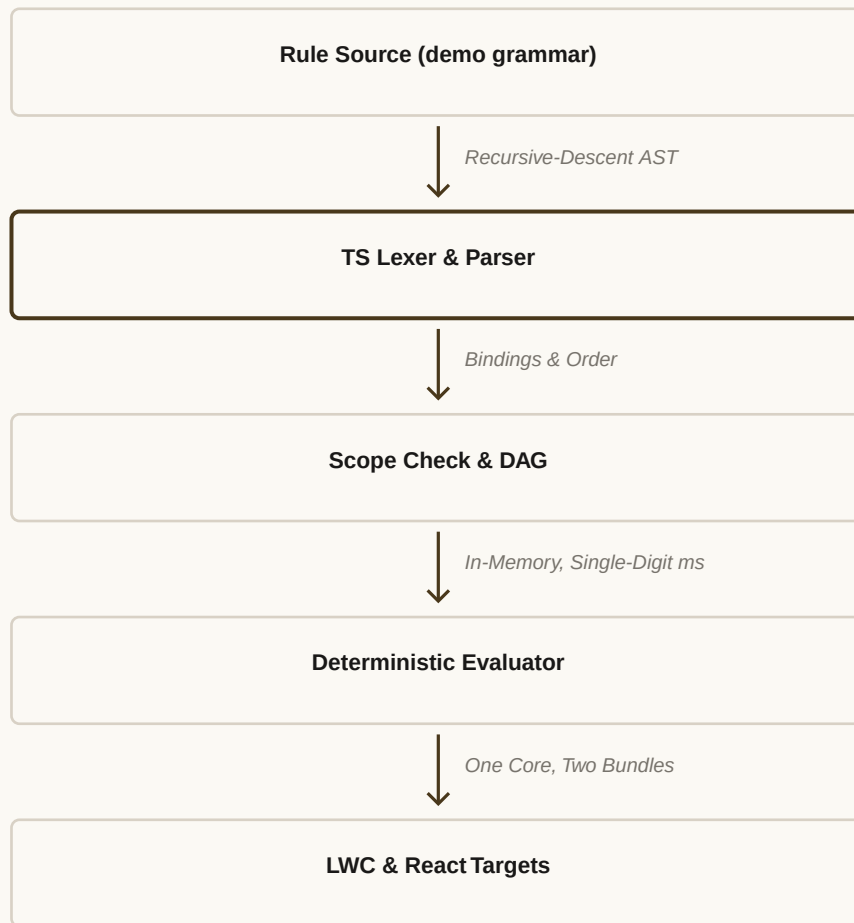
03 · THE ARCHITECTURE

Inspect the system

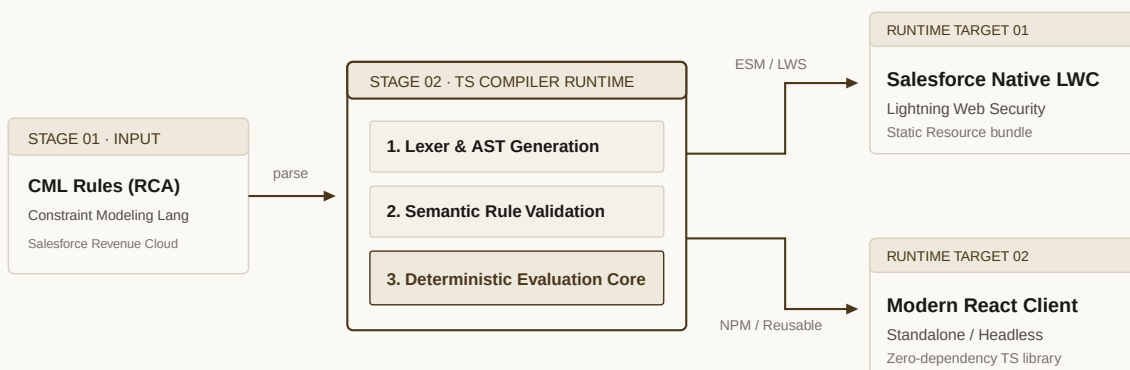
From rule text to evaluated result through a real language pipeline.

Four stages, one grammar, two runtimes, zero generated code.

CONSTRAINT COMPILER & DETERMINISTIC EVALUATION



Declarative rules compile to a typed AST once, then evaluate in memory on either runtime - no per-click server calls.



Full system schematic - Constraint Compiler & Deterministic Evaluation.

Boundary 01 / 06

Rule Source

Declarative constraints as text.

PURPOSE

Hold product logic in readable, versionable rule text instead of scattered conditionals.

BOUNDARY

It declares intent. It computes nothing and trusts nothing about runtimes.

FAILURE PREVENTED

Buried business rules that only one departed developer understood.

Boundary 02 / 06

Lexer & Parser

Hand-written, zero dependencies.

PURPOSE

Turn rule text into a typed abstract syntax tree with errors that point at the offending token.

BOUNDARY

Syntax only. It builds trees and reports errors; it knows no catalog and resolves no names.

FAILURE PREVENTED

Cryptic runtime misbehavior caused by silently misparsed rules.

Boundary 03 / 06

Scope Check & DAG

Bindings resolved, order computed.

PURPOSE

Resolve names against the catalog schema and order cascading attributes deterministically.

BOUNDARY

It plans execution order. It evaluates nothing.

FAILURE PREVENTED

Infinite re-evaluation loops and order-dependent misconfigurations.

Boundary 04 / 06

Deterministic Evaluator

Tree-walking, no code generation.

PURPOSE

Evaluate the compiled tree in memory with identical results on every runtime.

BOUNDARY

No eval, no generated functions, no network. Same input always yields the same output.

FAILURE PREVENTED

Sandbox rejections and the subtle evaluation drift of duplicated logic.

Boundary 05 / 06

Component Runtime Target

Sandbox-safe static bundle.

PURPOSE

Run the identical engine inside sandboxed CRM components.

BOUNDARY

It consumes the engine. Grammar and semantics stay defined once, upstream.

FAILURE PREVENTED

Per-click server latency and out-of-order response races in embedded UIs.

Boundary 06 / 06

Headless Storefront Target

Same engine, NPM import.

PURPOSE

Run the identical engine in headless storefronts with zero logic drift.

BOUNDARY

Presentation freedom above, identical arithmetic below.

FAILURE PREVENTED

Storefront-versus-CRM misquotes discovered months after deployment.

04 · FOLLOW THE FLOW

One rule, five hops

Watch a single constraint travel from text to truth value.

Now let's follow one rule from source text to evaluated result.

01 Rule Source

A product author writes a declarative constraint - eligibility, dependency, or pricing attribute - in reviewable text.

02 Lexer & Parser

The scanner and parser build a typed tree, modeling precedence and nesting. A syntax error points at its token here, not in production.

03 Scope Check & DAG

Names resolve against the catalog schema and cascading attributes sort into topological order. Cycles fail loudly at load time.

04 Deterministic Evaluator

The tree evaluates in memory in low single-digit milliseconds, with no code generation and no network involved.

05 Component Runtime Target

Inside the sandboxed component, the same engine answers the click instantly - no round-trip, no race.

06 Headless Storefront Target

On the headless storefront, the same engine answers the same input identically. The differential harness proves it, not promises it.

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.

Client-side evaluation core vs. per-click server calls

The default I saw everywhere was remote evaluation per interaction - correct, but slow and racy.

CHOSEN

A real compiler core in dependency-free TypeScript, evaluated in memory.

WHY

The network was the budget, not the CPU. Moving evaluation client-side removed the round-trip entirely; keeping it a proper AST (not string hacks) kept semantics identical across runtimes.

ALTERNATIVES CONSIDERED

- Server-side evaluators per keystroke
- Remote evaluation REST API
- Regex-based client parsing

Tree-walking interpreter vs. generating JS functions

`new Function()` would be faster on paper, but LWS and enterprise CSP forbid it - and rightly so.

CHOSEN

AST interpreter with deterministic coercion and node memoisation.

WHY

Compliance first, then speed. Short-circuit booleans and memoised subtrees kept my test models in the low single-digit milliseconds, which is fast enough that nobody notices the interpreter.

ALTERNATIVES CONSIDERED

- Dynamic `eval()` code generation
- Server-compiled bytecode cache

One shared library vs. Apex-plus-JavaScript dual maintenance

Every team I watched tried the dual implementation at least once. Every one of them paid for it in misquotes.

CHOSEN

A single canonical TS library consumed by both LWC and React.

WHY

One grammar, one parser, one evaluator. Drift becomes structurally impossible rather than process-prevented.

ALTERNATIVES CONSIDERED

- Hand-maintained storefront re-implementation
- Iframing the CRM UI
- Proxying storefront clicks through a gateway

Topological DAG resolution vs. multi-pass dirty-checking

Cascading attributes loop forever under naive re-evaluation.

CHOSEN

DAG with topological order and cycle detection at load time.

WHY

Single pass, deterministic order, and circular rules fail loudly at authoring time instead of freezing a UI thread.

ALTERNATIVES CONSIDERED

- Bounded multi-pass loops
- Chained event listeners

CHALLENGE THE DESIGN

Rules could compile to JavaScript functions for maximum speed. Why not generate code?

GENERATE FUNCTIONS WITH NEW FUNCTION()

Fastest per evaluation - and rejected at load by every sandboxed runtime and strict content policy that matters. Speed you cannot ship is not speed.

COMPILE ON THE SERVER, CACHE BYTECODE ON THE CLIENT

Keeps the client fast but reintroduces the server into every interaction: cache invalidation, versioning, and the round-trips you set out to kill.

INTERPRET THE TREE IN MEMORY

This is the architecture's answer. Compliant everywhere by construction, deterministic by design, and fast enough that the network saving dwarfs the overhead.

HOW THE ARCHITECT REASONED

Compliance first, then speed. Memoized subtrees and short-circuit evaluation keep realistic models in single-digit milliseconds.

The CRM team and the storefront team each want to own their rule logic. Allow it?

LET EACH TEAM OWN ITS COPY

Autonomy now, misquotes later. Two implementations of pricing arithmetic diverge the way rivers diverge - inevitably and silently.

PROXY STOREFRONT CLICKS THROUGH THE CRM

One implementation, but every interaction pays the network tax again - the original disease with extra infrastructure.

ONE ENGINE, TWO BUNDLES

This is the architecture's answer. Drift becomes structurally impossible rather than process-prevented, and both surfaces stay instant.

HOW THE ARCHITECT REASONED

A differential harness running thousands of cases through both bundles is the parity guarantee. Convention fades; structure holds.

06 · FAILURE MODES

Let's try to break it

Cycles, drift, and sandbox rejections - each with a landing place.

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

The circular rule

WHAT GOES WRONG

Attribute A depends on B, B depends on C, and C depends on A.

WHERE IT LANDS

Cycle detection at load time names the offending edge. The rule never reaches a UI thread to freeze.

The storefront misquote

WHAT GOES WRONG

A pricing rule changes and one surface computes differently than the other.

WHERE IT LANDS

Impossible by construction with one engine - and the differential harness would catch it before deployment if it ever became possible.

The sandbox rejection

WHAT GOES WRONG

A deployment target tightens its script policy against dynamic code.

WHERE IT LANDS

There is no dynamic code to reject. The tree-walking evaluator loads unchanged under stricter policies.

The 2am authoring error

WHAT GOES WRONG

A rule author ships a malformed expression on a Friday evening.

WHERE IT LANDS

The parser rejects it with a token-pointing error, and the build-time linter flags dead constraints and unknown attributes first.

The furious clicker

WHAT GOES WRONG

A user toggles options faster than any server could answer.

WHERE IT LANDS

Every evaluation is synchronous in-memory work. There are no responses to arrive out of order because nothing leaves the browser.

STANDING COMPROMISES

Bundle weight vs. network chatter

Gain: Interaction evaluation happens in memory with no network at all.

Cost: The client downloads and parses a small compiler library up front.

Mitigation: Kept it dependency-free and async-loadable; on my test models it parses in well under a frame.

Bounded grammar vs. full macro language

Gain: Evaluation time stays predictable - no runaway recursion in the UI thread.

Cost: No unbounded recursive macros in the rule language.

Mitigation: Everything I actually needed in configurators fits declarative constraints; macros were never worth the risk.

Interpreter overhead vs. JIT-compiled functions

Gain: Fully sandbox- and CSP-safe with no dynamic code.

Cost: Tree-walking is slower than native JIT per node.

Mitigation: Memoisation plus short-circuiting keeps realistic models in the low milliseconds - the network saving dwarfs it.

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 engine is strongest where rules are declarative and bounded: configuration, eligibility, pricing attributes. A real grammar with real errors beats clever string handling every time the catalog grows.

Its limits are stated, not hidden. Unbounded macros are deliberately excluded, the grammar evolves by hand, and parity rests on the harness being run - a guarantee that only works if it stays in CI.

WHAT I WOULD CHANGE TODAY

- Explore a Rust port of the evaluator behind the identical grammar and differential tests
- Add error recovery to the parser so one bad rule never hides the next

- Publish the differential harness as the primary artifact, engine second

VERIFIED OUTCOMES

- Interactive evaluation moved from per-click server latency to in-memory execution (low single-digit ms across my differential harness models)
- One rule implementation serves both LWC and React paths - verified by differential tests, not by convention
- Authoring errors surface at build time (unknown attributes, cycles, dead rules) instead of as production misquotes
- Sandbox-safe by construction: zero dynamic code generation, loads under LWS and strict CSP
- Reusable across demo models without modification - the engine knows nothing about any employer's catalog

Continue the exploration

[Interactive exhibit: victorkuldeep.com/case-studies/cml-compiler/deep-dive](https://victorkuldeep.com/case-studies/cml-compiler/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





CASE STUDY 03 · DEEP DIVE EXHIBIT · ~8 MIN

Generic Entity Scoring Engine, Native to the CRM

“The hard problem is not adding an agent. It is deciding which agent - and whether any agent - should act.”

ROLE

Independent AI Systems Researcher

TIMELINE

Independent research · 2025–2026

DOMAIN

Agentic Systems & AI Architecture

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Independent research prototype built on synthetic CRM-like fixtures - no production records, no employer tooling, no client workflows. I prototyped a generic scoring engine that grades the readiness of any record type with the same deterministic algorithm - completeness, urgency, operational risk - then routes qualifying work to one narrow specialist with a fresh context window, and executes every mutating tool behind JSON-schema validation with an audit envelope. The pattern under study is CRM-adjacent (approvals, escalations, provisioning-style flows) but the fixtures are mine and the code is standalone. Nothing here describes any employer's internal systems.

Independent prototype, synthetic data. I built this because single-agent demos kept failing the same way in my experiments: one bloated prompt, thirty tools, and no notion of whether the record was even ready. The engine is deliberately generic - the same scoring algorithm runs unchanged across different fixture entities, so readiness means the same thing everywhere instead of being redefined per object. I wanted that boring machinery - scoring, routing, gating - treated as first-class architecture instead of prompt tricks.

What goes wrong without it: models get invoked on incomplete records, contexts bloat across domains, tools fire with unvalidated arguments, and nobody can reconstruct why an action happened. In record-centric work that is unacceptable.

CONSTRAINTS

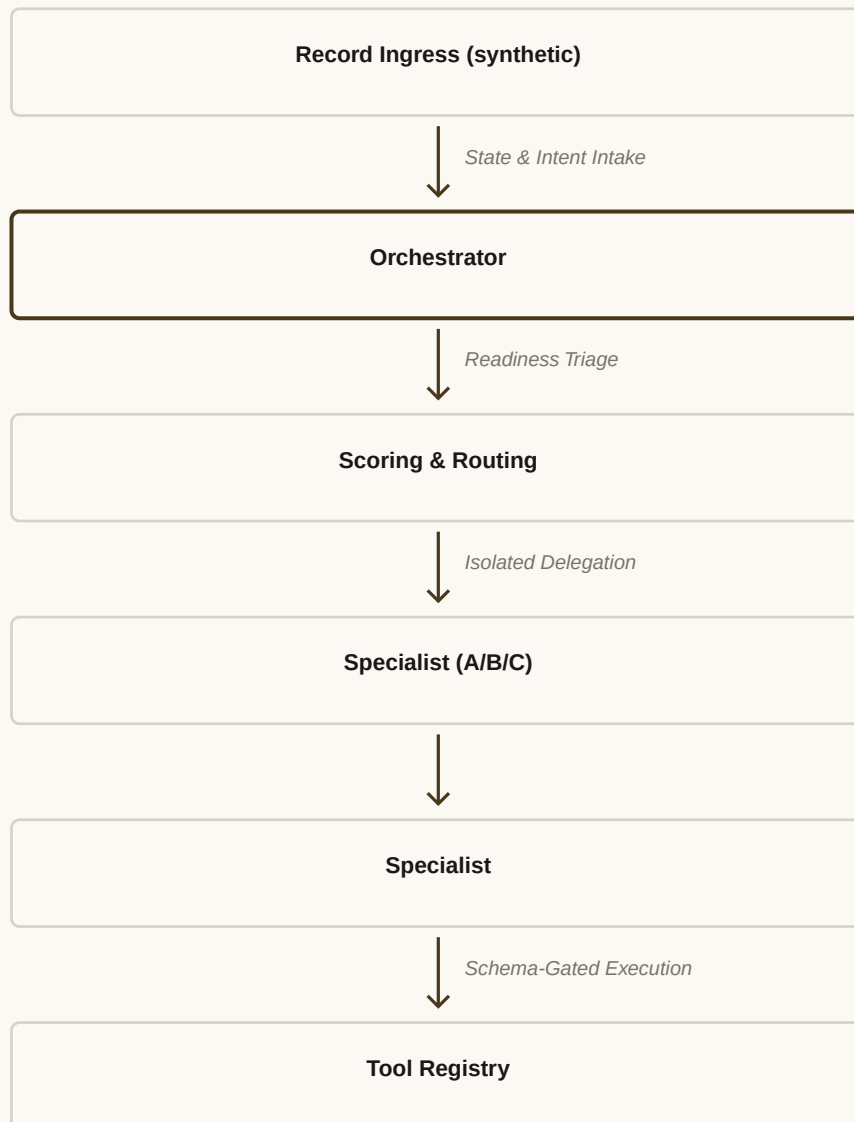
- No model call on unready records - deterministic triage first
- Every mutation behind schema validation and permission checks
- One specialist per task with an isolated context window
- Full reconstruction: scores, routes, and tool I/O all auditable

PART I · ARCHITECTURE

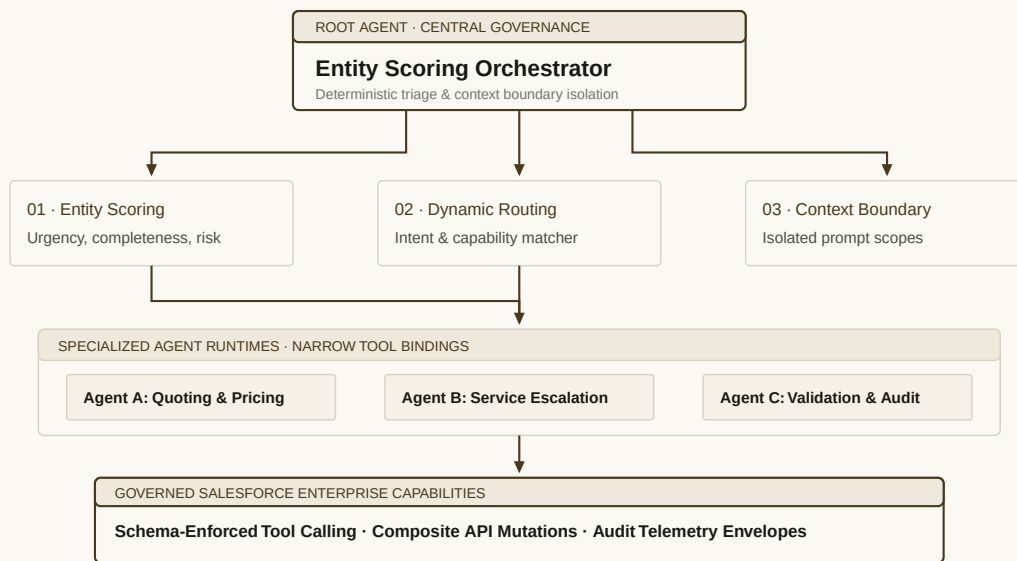
The system as designed

Three stages, deliberately unglamorous: 1) Score - completeness, urgency, and risk computed deterministically over the fixture record. Below threshold, the orchestrator stops and says why. 2) Route - an intent classifier picks exactly one specialist and hands it a fresh, narrow context. 3) Act - the specialist may only call registered tools whose arguments validate against strict schemas; every call and result lands in an audit envelope.

SCORE-FIRST ORCHESTRATOR & SPECIALIST ROUTING



Triage before tokens: score, route to one specialist, and gate every tool call behind schemas.



Full system schematic - Score-First Orchestrator & Specialist Routing.

PART I · DECISIONS

Choices, engineering, tradeoffs

Score-first triage vs. immediate generation

Calling a model on a half-formed record wastes tokens and invites confident nonsense.

CHOSEN

Deterministic scoring gate before any model invocation.

WHY

Cheap arithmetic filters most cases; expensive inference runs only where it can help. My logs showed fewer wasted calls and clearer refusals.

Narrow specialists vs. one mega-prompt

One agent holding quoting, SLA, and billing logic degrades at all three.

CHOSEN

Central router with small, single-domain specialists.

WHY

Small contexts stay sharp. Routing decisions stay visible. Debugging stops being archaeology.

ENGINEERING NOTES

- Built the scorer as pure functions over fixture records - no model involved, fully unit-testable thresholds.
- Implemented the router as intent classification plus capability matching, with explicit abstention when nothing qualifies.
- Registered every tool with a strict JSON schema; validation failures return structured errors, never partial writes.
- Wrapped each run in an audit envelope: input hash, scores, route taken, tool calls, outputs, timestamps.
- Gave each specialist a fresh context assembled from the envelope - never the full conversation history.

PART I · OUTCOMES

Tradeoffs, results, lessons

Routing hops vs. single-shot latency

Gain: Guardrails, visibility, and far fewer runaway actions.

Cost: Extra milliseconds per run for scoring and routing.

Mitigation: Precomputed score vectors and parallel skill-eligibility checks keep the overhead small.

Schema gates vs. freeform tools

Gain: Invalid mutations die at the boundary with a reason attached.

Cost: Each capability needs a written schema and mapping.

Mitigation: A small tool template keeps new capabilities to one file each.

OUTCOMES

- Unready fixtures stop before any model call, with a machine-readable reason
- Specialist contexts stay small and task-shaped across multi-step fixture runs
- Schema gates catch malformed tool arguments before any state change
- Every run reconstructs end to end from its audit envelope

LESSONS

- Score before you generate; most enterprise records are not ready on arrival.
- Small routed contexts outperform large shared ones in every run I measured.

01 · ORIENTATION

An exhibit, not an article

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

You are not just reading about this architecture. You are going to inspect how it works.

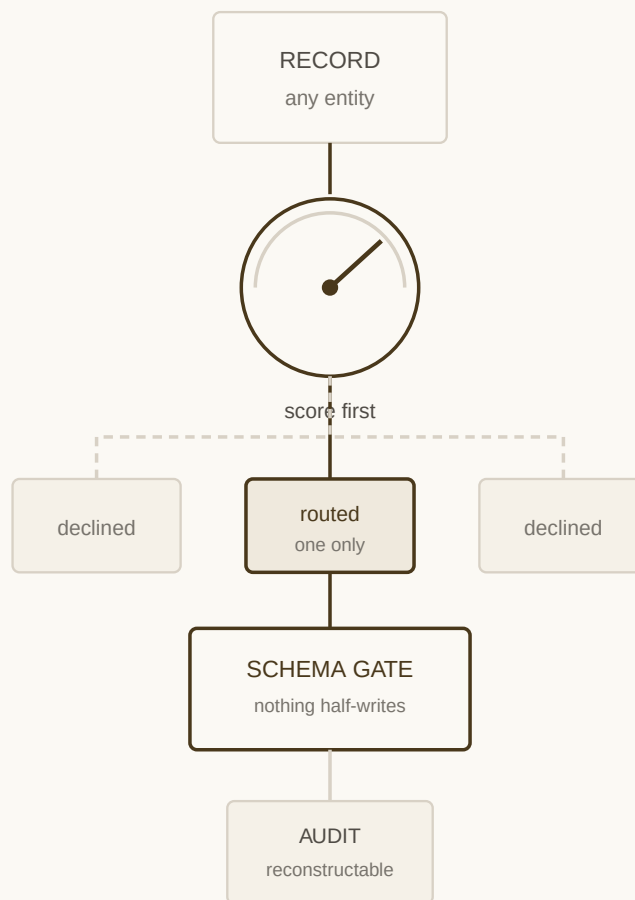


Exhibit plate - the system at a glance.

This exhibit walks through a generic entity scoring engine: one deterministic algorithm that grades any record type, a router that picks exactly one specialist, and schema gates around every mutation. Six short chapters, each with something to touch.

Nothing here is a production system. The fixtures are synthetic, the runs are described rather than executed, and every number comes from the author's own test harness. What is real is the reasoning: why each boundary exists and which failure it absorbs.

How to use this exhibit

- Inspect any component to see its purpose, boundary, and the failure it prevents
- Step one request through the system hop by hop
- Answer two design questions before reading how the architect reasoned
- Try to break the system, then see where each failure lands

02 · THE PROBLEM

Start from the failure

Every boundary in this system answers a failure. Meet the failures first.

Before we look at the architecture, consider the failure it exists to prevent.

Single-agent demos fail the same way every time: one bloated prompt holding quoting logic, service policy, and billing rules at once, thirty tools attached, and no notion of whether the record is even ready for automated handling.

The consequences stack up. Models get invoked on incomplete records, burning tokens to produce confident nonsense. Contexts bloat across domains until nothing is sharp. Tools fire with unvalidated arguments against real records. And when someone asks why an action happened, there is nothing to point at.

In record-centric work that is unacceptable. The full narrative lives on the case study page - here we start from that failure and build outward.

THE CONSTRAINTS THIS FAILURE IMPOSES

- No model call on unready records - deterministic triage first
- Every mutation behind schema validation and permission checks
- One specialist per task with an isolated context window
- Full reconstruction: scores, routes, and tool I/O all auditable

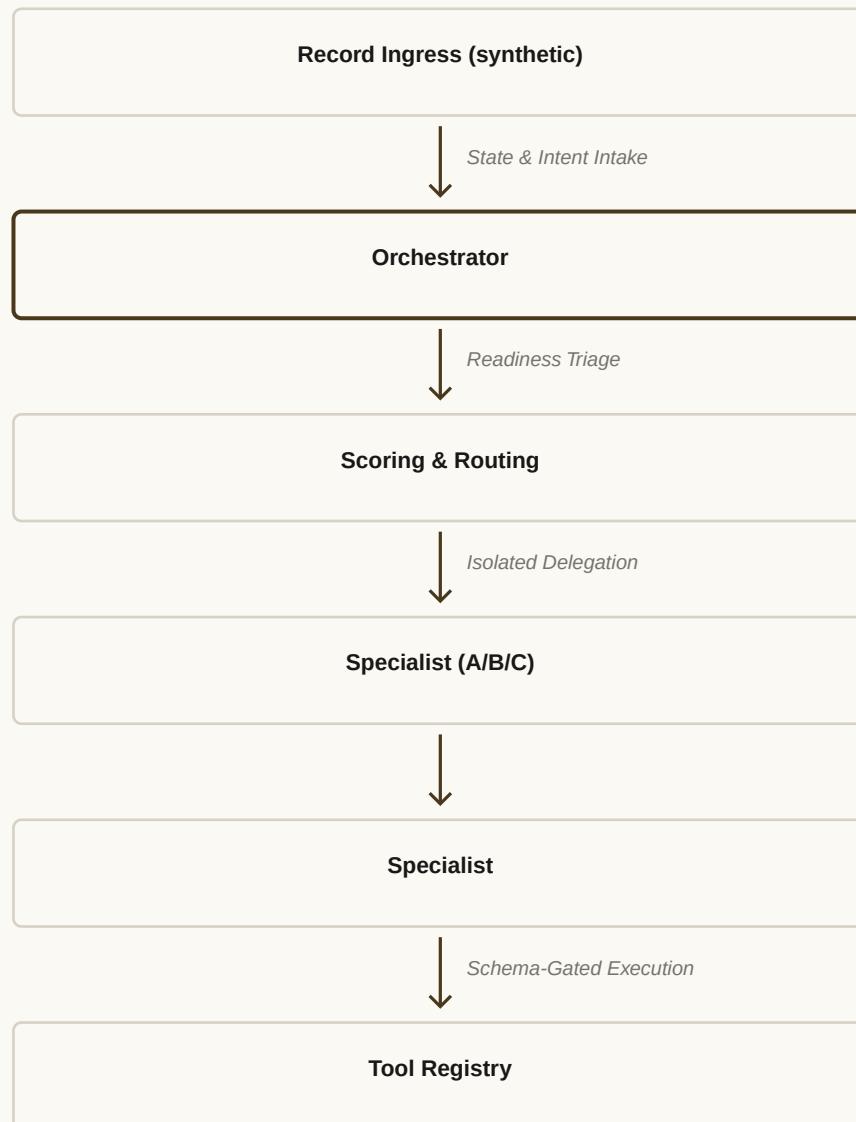
03 · THE ARCHITECTURE

Inspect the system

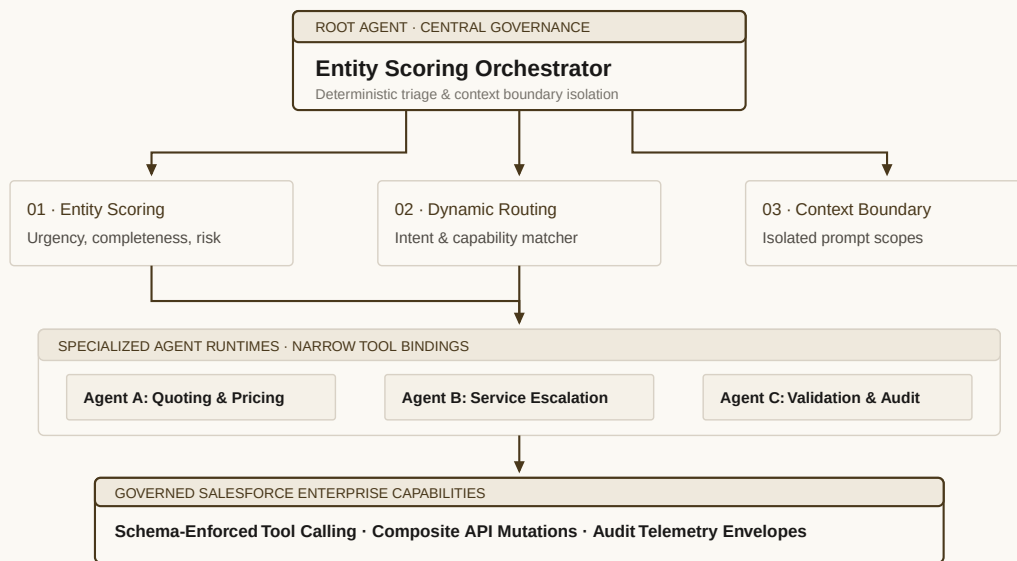
Explore the boundaries that make the architecture work.

Six components. Each one exists because a specific failure demanded it.

SCORE-FIRST ORCHESTRATOR & SPECIALIST ROUTING



Triage before tokens: score, route to one specialist, and gate every tool call behind schemas.



Full system schematic - Score-First Orchestrator & Specialist Routing.

Boundary 01 / 06

Entity Ingress

Capture & normalize entity data.

PURPOSE

Normalize an incoming request and its fixture record into the shape the engine reasons about.

BOUNDARY

It transforms and forwards. It never judges readiness and never invokes a model.

FAILURE PREVENTED

Format drift across entity types, where each stage quietly assumes a different record shape.

Boundary 02 / 06

Entity Intelligence

Score, enrich & assess readiness.

PURPOSE

Grade record readiness with pure functions before a single token is spent.

BOUNDARY

Arithmetic only. No model calls, no judgment calls, no side effects - it scores and steps aside.

FAILURE PREVENTED

Confident model output on half-formed records, plus the token spend that comes with it.

Boundary 03 / 06

Intent Router

Decide the right capability.

PURPOSE

Pick exactly one specialist for qualified work - or decline the work explicitly.

BOUNDARY

It routes. It never executes domain work itself and never fans out to several specialists at once.

FAILURE PREVENTED

Silent misrouting, where the wrong agent acts confidently on a task it half-understands.

Boundary 04 / 06

Task-Specialized Agent

Execute with guardrails.

PURPOSE

Do one domain's work with a fresh, tightly scoped context.

BOUNDARY

One domain per specialist. It never sees the full conversation history and never calls tools outside its registry.

FAILURE PREVENTED

Context bloat and cross-domain hallucination, where quoting logic leaks into billing actions.

Boundary 05 / 06

Governed Tool Boundary

Controlled access to enterprise tools.

PURPOSE

Stand between specialist intent and state change, and reject anything malformed.

BOUNDARY

Validation failures return structured errors. Nothing half-writes, ever.

FAILURE PREVENTED

Invalid mutations reaching the record store - wrong types, missing fields, out-of-scope actions.

Boundary 06 / 06

Audit Envelope

Trace, evaluate and learn.

PURPOSE

Make every run reconstructable long after it finishes.

BOUNDARY

It observes and records. It never alters execution or hides fields for convenience.

FAILURE PREVENTED

The unanswerable 'why did it do that' - the failure mode that kills automation programs quietly.

04 · FOLLOW THE FLOW

One request, six hops

Watch a single request earn its way through, hop by hop.

Now let's follow one request through the system, one hop at a time.

01 Entity Ingress

A request arrives with a fixture record. Nothing is assumed about completeness - the envelope just normalizes the shape.

02 Entity Intelligence

Pure functions grade completeness, urgency, and risk. Below threshold the run stops here with a machine-readable reason, and no model is ever invoked.

03 Intent Router

Intent classification plus capability matching selects exactly one specialist. If nothing qualifies, the orchestrator abstains rather than guessing.

04 Task-Specialized Agent

The specialist gets a fresh, narrow context assembled from the envelope - never the full history - and plans its tool calls.

05 Governed Tool Boundary

Each requested call validates arguments against its JSON schema and permission scope. Failures return structured errors; nothing half-writes.

06 Audit Envelope

Input hash, scores, route, tool I/O, and timestamps land in the envelope. The run now reconstructs end to end, days later if needed.

05 · ARCHITECTURE DECISIONS

The choices that hold it up

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

You have seen the topology. Here are the two decisions that hold it up - each with the alternative it beat.

Score-first triage vs. immediate generation

Calling a model on a half-formed record wastes tokens and invites confident nonsense.

CHOSEN

Deterministic scoring gate before any model invocation.

WHY

Cheap arithmetic filters most cases; expensive inference runs only where it can help. My logs showed fewer wasted calls and clearer refusals.

ALTERNATIVES CONSIDERED

- Prompt directly on the raw record
- Hardcoded triggers with no model at all

Narrow specialists vs. one mega-prompt

One agent holding quoting, SLA, and billing logic degrades at all three.

CHOSEN

Central router with small, single-domain specialists.

WHY

Small contexts stay sharp. Routing decisions stay visible. Debugging stops being archaeology.

ALTERNATIVES CONSIDERED

- Single prompt with 30+ tools
- Uncoordinated agent swarm

CHALLENGE THE DESIGN

A new request arrives on a half-complete record. What is your first move?

INVOKE THE MODEL IMMEDIATELY

Tempting - the model is the powerful part. But on a half-formed record it spends tokens to produce confident nonsense, and you cannot distinguish its guesses from knowledge.

APPLY HARDCODED RULES, NO MODEL AT ALL

Safe and cheap, until the rules meet a record they were not written for. Pure rules do not degrade gracefully; they just stop covering reality.

SCORE THE RECORD FIRST, THEN DECIDE

This is the architecture's answer. Cheap arithmetic filters most cases and states its reason; expensive inference runs only where it can actually help.

HOW THE ARCHITECT REASONED

The logs settled it: scoring first cut wasted model calls and turned silent guesses into explicit refusals with reasons attached. Triage before tokens.

One agent could hold quoting, service policy, and billing knowledge. Ship it?

SHIP THE SINGLE MEGA-PROMPT

One prompt, thirty tools, three domains. It degrades at all three simultaneously, and debugging means archaeology through one giant context.

LET UNCOORDINATED AGENTS COLLABORATE

Autonomy without a router is just several agents guessing at once. Nobody owns the route, so nobody can explain it later.

ROUTE TO NARROW SPECIALISTS

This is the architecture's answer. Small contexts stay sharp, routing decisions stay visible, and each specialist is boring enough to trust.

HOW THE ARCHITECT REASONED

A central router with single-domain specialists confines each context window to one job. The routing decision is recorded, so precision and observability arrive together.

06 · FAILURE MODES

Let's try to break it

Five failures, five landing places. That is the design working.

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

The half-complete record

WHAT GOES WRONG

A request arrives below the readiness threshold - missing fields, stale state, ambiguous intent.

WHERE IT LANDS

The scorer stops the run before any model call and returns a machine-readable reason. Cost of the failure: arithmetic.

The task that spans three domains

WHAT GOES WRONG

A request touches quoting, service policy, and billing in one breath - the shape that bloats a shared context.

WHERE IT LANDS

The router isolates it to one specialist with a fresh, narrow context. The other domains never enter the window.

The wrong specialist

WHAT GOES WRONG

Classification confidence is low, or the chosen specialist cannot cover the task.

WHERE IT LANDS

The orchestrator abstains with a stated reason, and the audit envelope records the route that was considered - misroutes become visible, reviewable events.

The malformed tool call

WHAT GOES WRONG

A specialist emits tool arguments with wrong types, missing fields, or out-of-scope actions.

WHERE IT LANDS

The schema gate rejects the call with a structured error before any state change. Nothing half-writes, ever.

The unanswerable 'why did it do that'

WHAT GOES WRONG

Days later, someone must reconstruct a run - for debugging, review, or accountability.

WHERE IT LANDS

The audit envelope replays input hash, scores, route, and tool I/O in order. Reconstruction is a read operation, not an investigation.

STANDING COMPROMISES

Routing hops vs. single-shot latency

Gain: Guardrails, visibility, and far fewer runaway actions.

Cost: Extra milliseconds per run for scoring and routing.

Mitigation: Precomputed score vectors and parallel skill-eligibility checks keep the overhead small.

Schema gates vs. freeform tools

Gain: Invalid mutations die at the boundary with a reason attached.

Cost: Each capability needs a written schema and mapping.

Mitigation: A small tool template keeps new capabilities to one file each.

07 · REFLECTION

What remains true

Limits, stated plainly - and what changes next.

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

This engine is strongest where the world is legible: records with definable completeness, tasks with classifiable intent, tools with writable schemas. Where those hold, the boring machinery outperforms clever prompting every time.

Its limits are stated, not hidden. Thresholds are hand-tuned on fixtures, runs are single-request rather than conversational, and the router classifies rather than retrieves. Each limit is a named next experiment, not a surprise.

WHAT I WOULD CHANGE TODAY

- Learn score thresholds from labeled fixture outcomes instead of hand-tuning them
- Retrieve over capability documentation so the tool registry scales past hand-written entries
- Extend the envelope across multi-turn runs without letting history leak back into specialist contexts

VERIFIED OUTCOMES

- Unready fixtures stop before any model call, with a machine-readable reason
- Specialist contexts stay small and task-shaped across multi-step fixture runs
- Schema gates catch malformed tool arguments before any state change
- Every run reconstructs end to end from its audit envelope

Continue the exploration

[Interactive exhibit: victorkuldeep.com/case-studies/scoring-intelligence-agent/deep-dive](https://victorkuldeep.com/case-studies/scoring-intelligence-agent/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



KULDEEP SINGH



CASE STUDY 04 · DEEP DIVE EXHIBIT · ~6 MIN

TM Forum Open API Intelligence Suite & Spectra AI Copilot

“When enterprise JSON payloads become unreadable, transform data contracts into observable, interactive graphs with client-side AI.”

ROLE

Creator & Principal Systems Architect

TIMELINE

Architecture & Open Systems Initiative · 2026

DOMAIN

Telco Systems & Applied AI Architecture

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Architected a browser-local enterprise suite for 23+ TM Forum Open APIs. Combines React Flow DAG visualization, an interactive laser presentation surface, AJV schema compliance, and Spectra AI - a hybrid AI orchestrator pairing in-browser WebGPU local inference (Qwen 3.5) with direct-CORS cloud models.

Modern telecommunications architectures undergoing TM Forum Open Digital Architecture (ODA) transformations rely on over 23 Open APIs (TMF620 Product Catalog, TMF622 Product Ordering, TMF637 Product Inventory, TMF641 Service Ordering, TMF679 Offering Qualification, and related domains). Enterprise payloads in these systems frequently span multi-megabyte, deeply nested JSON documents featuring multi-level product hierarchies, dynamic characteristics, place relationships, and complex pricing elements. Standard text editors and tree viewers become virtually unusable, leading to severe communication friction during architectural reviews and client discovery workshops.

Telco architects, BSS engineers, and solution teams lacked a dedicated visual and analytical medium to inspect, present, and validate complex TMF payloads. Existing generic JSON tools offer no semantic understanding of TM Forum data models, collapse under circular references, offer no schema compliance diagnostics, and lack real-time presentation capabilities. Furthermore, strict enterprise data privacy rules prevent teams from sending sensitive client telecom contracts and quote payloads to third-party cloud AI services for analysis.

CONSTRAINTS

- Must run 100% client-side in the browser with zero mandatory backend server dependencies to protect proprietary enterprise telco data
- Sub-second DAG graph rendering and layout for massive hierarchical order and inventory payloads (up to 10MB JSON)
- Spectra AI must support local in-browser WebGPU inference (zero token cost, zero data leakage) alongside native direct CORS cloud models
- Strict schema validation supporting AJV Draft-07 and 2020-12 across 23+ TM Forum Open API specifications with path-level error diagnostics
- Seamless real-time presentation mode with laser pointer drawing trails to facilitate live architectural brainstorming and client discovery

PART I · ARCHITECTURE

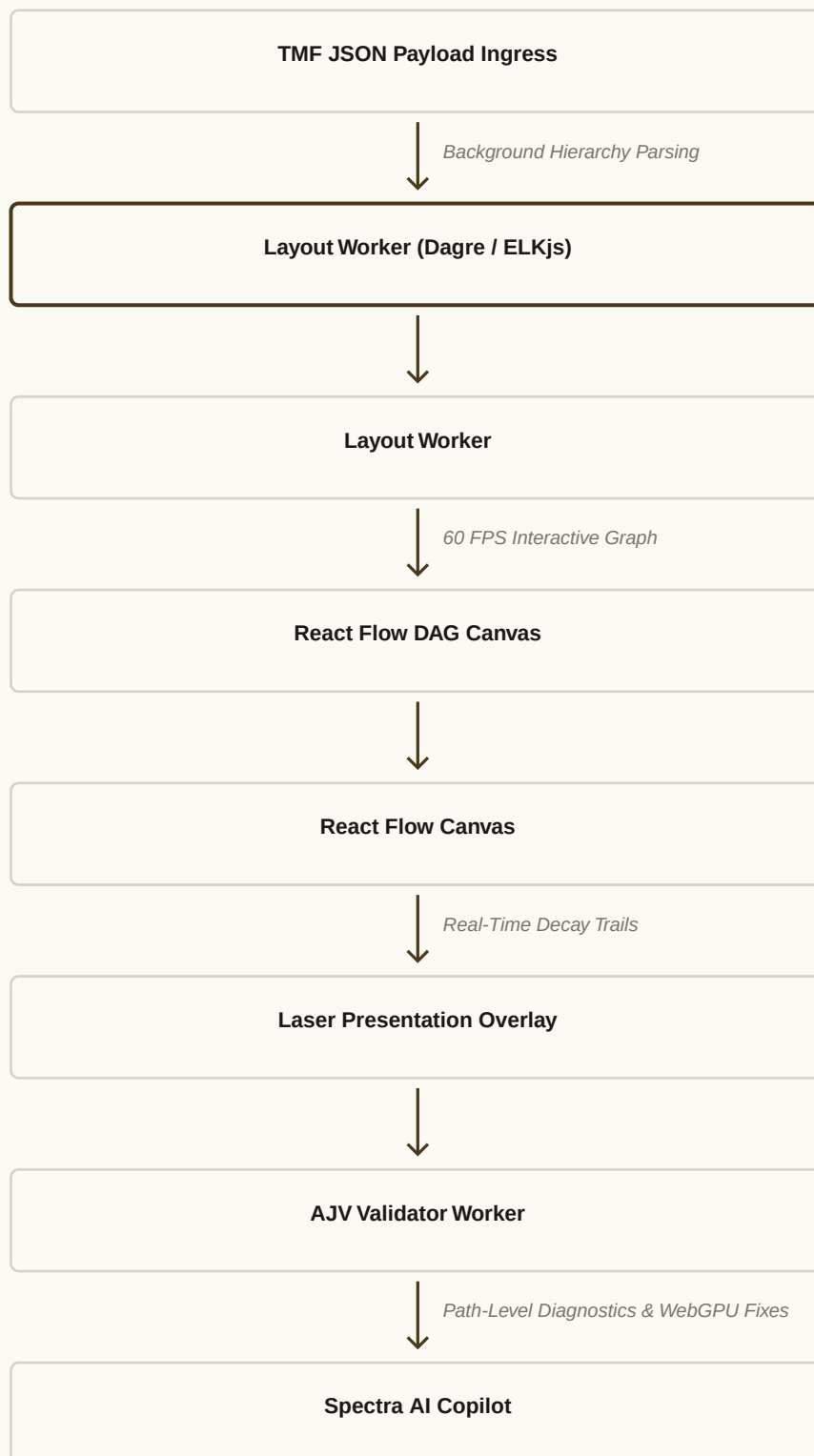
The system as designed

Architected as an ultra-fast, client-side Single Page Application (SPA) pairing React Flow (@xyflow/react) with asynchronous graph layout worker threads (Dagre and ELKjs). Built an interactive presentation overlay with mathematical laser trail decay, a real-time schema diffing engine, and an AJV-powered validator. The intelligence layer, Spectra AI Copilot, is engineered as a hybrid AI orchestrator: users can sign in with Google to execute

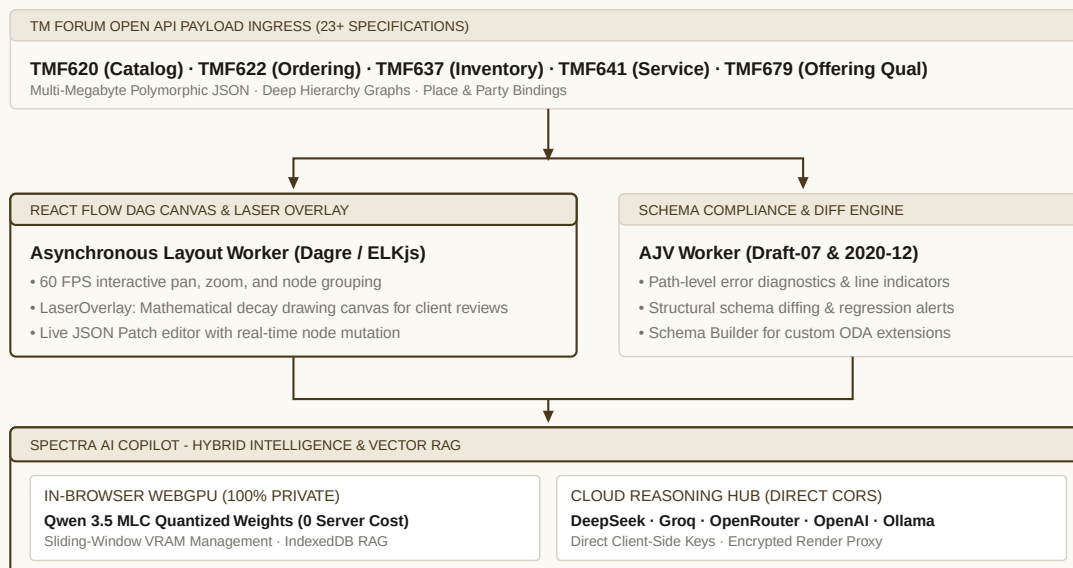
local quantized LLMs (Qwen 3.5 builds) directly on their client GPU via browser WebGPU, or connect directly to cloud reasoning models (DeepSeek, Groq, OpenRouter, OpenAI, and Ollama) with zero backend retention. An IndexedDB vector store enables local RAG over custom user guides with exact specification citations.

KULDEEP SINGH

TM FORUM API VISUALIZER & SPECTRA AI COPILOT ARCHITECTURE



Client-side single-page architecture orchestrating React Flow DAG layout workers, SVG laser drawing canvas, AJV schema compliance, and hybrid WebGPU/Cloud Spectra AI inference.



Full system schematic - TM Forum API Visualizer & Spectra AI Copilot Architecture.

PART I · DECISIONS

Choices, engineering, tradeoffs

React Flow & Directed Acyclic Graph (DAG) Layouts vs. Static Collapsible JSON Trees

Nested JSON trees obscure topological relationships between parent/child order items, billing accounts, and qualifying characteristics.

CHOSEN

React Flow graph canvas orchestrated with Dagre and ELKjs layout engines inside dedicated Web Workers.

WHY

Transforms abstract data contracts into an intuitive visual graph where architects can navigate dependencies, inspect polymorphic attributes, and understand fulfillment topology in real time.

In-Browser WebGPU Local AI + Native CORS Cloud Hub vs. Centralized Heavy AI Server

Enterprise telco payloads contain commercially sensitive pricing, network topologies, and customer identities that cannot touch third-party servers.

CHOSEN

Hybrid AI architecture featuring in-browser WebGPU inference via @mlc-ai/web-llm (Qwen 3.5) paired with native browser CORS for cloud models.

WHY

Provides 100% data privacy with zero server compute costs. Prompts and weights run purely in local browser memory and WebGPU vRAM.

Real-Time Laser Pointer Presentation Canvas vs. Screen Annotation Plugins

Architects frequently conduct live virtual sessions to explain complex payload dependencies to stakeholders without screen clutter.

CHOSEN

Custom SVG pointer drawing surface (LaserOverlay) with fading quadratic bezier trails and 1500ms glow decay.

WHY

Allows presenters to naturally guide client attention directly on active nodes without leaving the browser or installing extra software.

AJV Schema Compliance & Live JSON Patching vs. Read-Only Static Validators

Architects not only need to find errors in payloads; they need to simulate fixes and author schema extensions on the fly.

CHOSEN

Integrated Schema Hub and Schema Builder with path-level JSON patch generation and instantaneous visual feedback.

WHY

Accelerates new TMF API development cycles by allowing engineers to modify payloads and verify compliance live.

ENGINEERING NOTES

- Decoupled layout computation into a dedicated Web Worker (layout.worker.js) utilizing Dagre and ELKjs, keeping the React UI thread at a smooth 60 FPS even when rendering hundreds of node hierarchies.
- Implemented a dynamic local sliding window memory algorithm for WebGPU inference that automatically scales context to protect browser GPU memory and prevent vRAM exhaustion.
- Engineered an SVG-based laser trail renderer with decay timers and pointer capture (laserTrail.js), enabling smooth multi-stroke drawing across zoomable canvas surfaces.
- Constructed a browser-local vector indexing pipeline (docChunker.js, pdfTextExtractor.js, vectorStore.js) using IndexedDB (spectra_chats_db) for local document RAG with API scope tagging.
- Built dual deployment pipelines: static production on Hostinger CDN (tmf.victorkuldeep.com) and Cloudflare Workers (tmf-visualizer.victorkuldeep.workers.dev) with custom worker routing.

PART I · OUTCOMES

Tradeoffs, results, lessons

In-Browser WebGPU Compute vs Cloud API Model Capabilities

Gain: Zero server hosting cost, zero token billing, and total compliance with strict enterprise data residency rules.

Cost: Requires client device with WebGPU support (Chrome/Edge/Brave) and initial download of quantized model weights (~500MB - 1.5GB).

Mitigation: Engineered a dual-engine Model Hub: users without WebGPU can connect lightweight cloud keys (DeepSeek/Groq) with native CORS or run local Ollama/LM Studio.

Full Client-Side Schema Validation vs Server-Side Verification

Gain: Instant sub-millisecond feedback on every keystroke without network latency.

Cost: Requires packaging schemas for 23 TM Forum APIs into client bundle chunks.

Mitigation: Employed dynamic schema registry caching and lazy-loading for API definitions (TMF620-TMF764).

OUTCOMES

- Deployed live to production at tmf.victorkuldeep.com and Cloudflare Workers UAT mirror with global sub-second availability
- Comprehensive visual, structural, and compliance support for 23 TM Forum Open APIs spanning Product, Service, Resource, Customer, and Billing
- Zero server token costs: 100% private in-browser WebGPU AI inference with Qwen 3.5 local models
- Adopted by architects and BSS engineers for interactive client design sessions using the built-in laser pointer presentation tool
- Accelerated schema compliance and payload debugging from hours of manual JSON parsing to instant visual diagnosis

LESSONS

- Data contracts should not be locked in raw text; visual topology reveals structural defects that syntax highlighting cannot.
- Client-side WebGPU and browser-local RAG represent the future of enterprise AI for confidential customer payloads.
- Building specialized, domain-tailored visual tools elevates architectural influence far beyond standard slide decks and static documentation.

01 · ORIENTATION

An exhibit, not an article

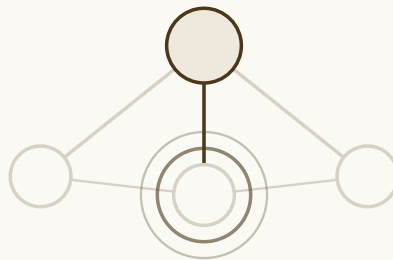
Seven layers, each with something to touch. Start anywhere - the path remembers where you are.

You are about to read a telecom payload the way architects wish they could - as a living diagram.

```
{ "product": {  
  "characteristics": [  
    { "place": {...}},  
    { "party": {...}},  
    ... 2.4 MB more ...  
  ]  
}
```

nested · unreadable

layout



topology reveals structure

STAYS ON THIS MACHINE

local inference · zero transit

Exhibit plate - the system at a glance.

This exhibit walks through a browser-local suite for 23+ TM Forum Open APIs: multi-megabyte JSON payloads become navigable graphs, a laser pointer draws over live topology, schemas validate path by path, and a hybrid AI copilot explains what you are looking at.

Everything runs client-side. That constraint is the whole design: telco payloads carry commercial sensitivity, so nothing may require a server to function.

How to use this exhibit

- Inspect each layer of the visualization pipeline
- Step one payload from raw JSON to presented insight
- Decide where AI inference should physically run
- Break the payload, then watch validation and layout respond

02 · THE PROBLEM

The unreadable contract

Megabyte payloads, no semantics, no presentation surface, and data that cannot travel.

Start from the unreadable payload, not the tooling.

TM Forum payloads span megabytes of nested product hierarchies, dynamic characteristics, and pricing elements. Generic JSON viewers collapse under circular references, say nothing about TM Forum semantics, and offer no way to present findings to a room.

Worse, the obvious AI answer - paste the payload into a cloud model - violates the data rules of every serious engagement. Sensitive contracts cannot leave the building to be understood.

THE CONSTRAINTS THIS FAILURE IMPOSES

- Must run 100% client-side in the browser with zero mandatory backend server dependencies to protect proprietary enterprise telco data
- Sub-second DAG graph rendering and layout for massive hierarchical order and inventory payloads (up to 10MB JSON)
- Spectra AI must support local in-browser WebGPU inference (zero token cost, zero data leakage) alongside native direct CORS cloud models
- Strict schema validation supporting AJV Draft-07 and 2020-12 across 23+ TM Forum Open API specifications with path-level error diagnostics
- Seamless real-time presentation mode with laser pointer drawing trails to facilitate live architectural brainstorming and client discovery

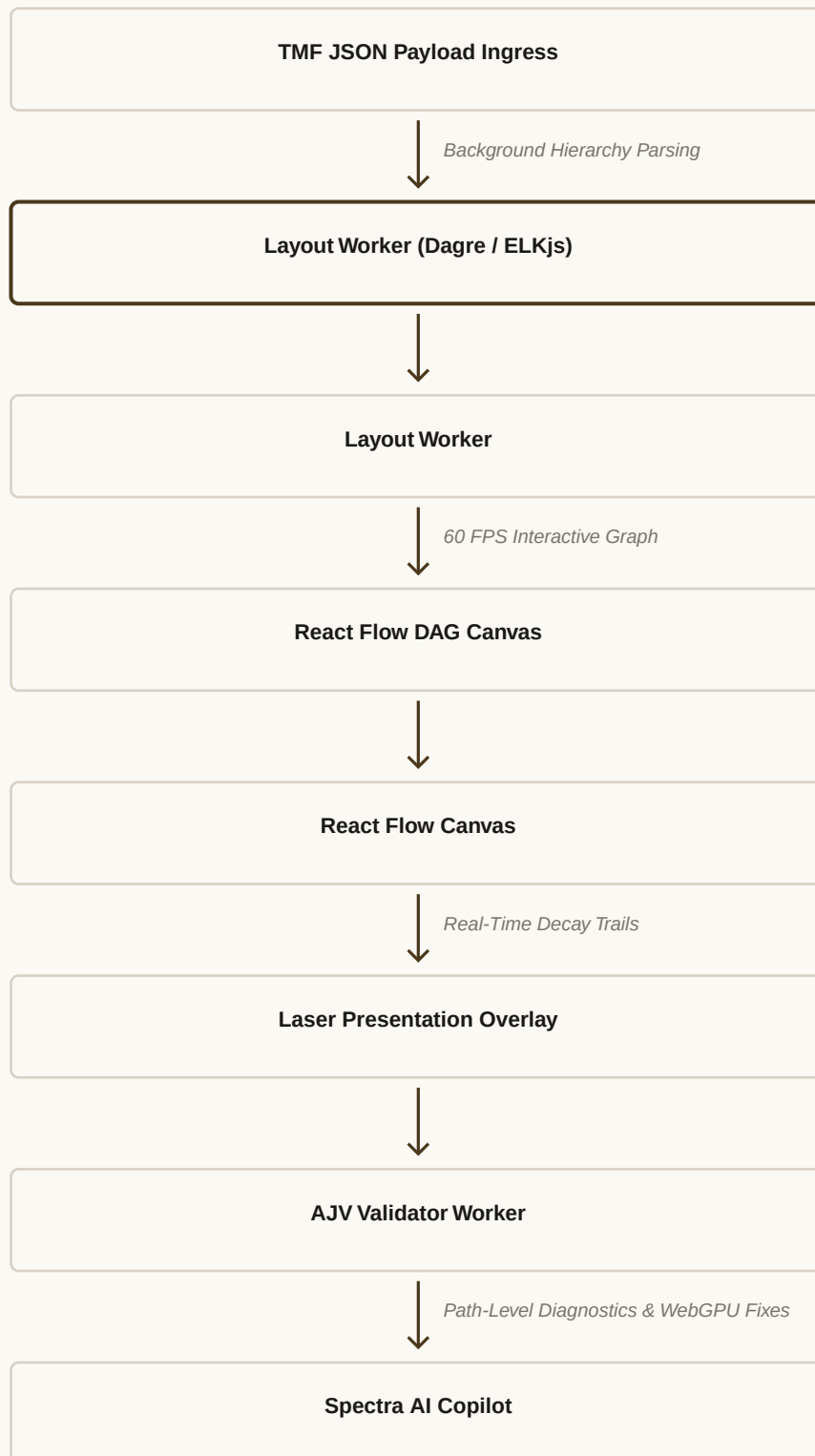
03 · THE ARCHITECTURE

Inspect the system

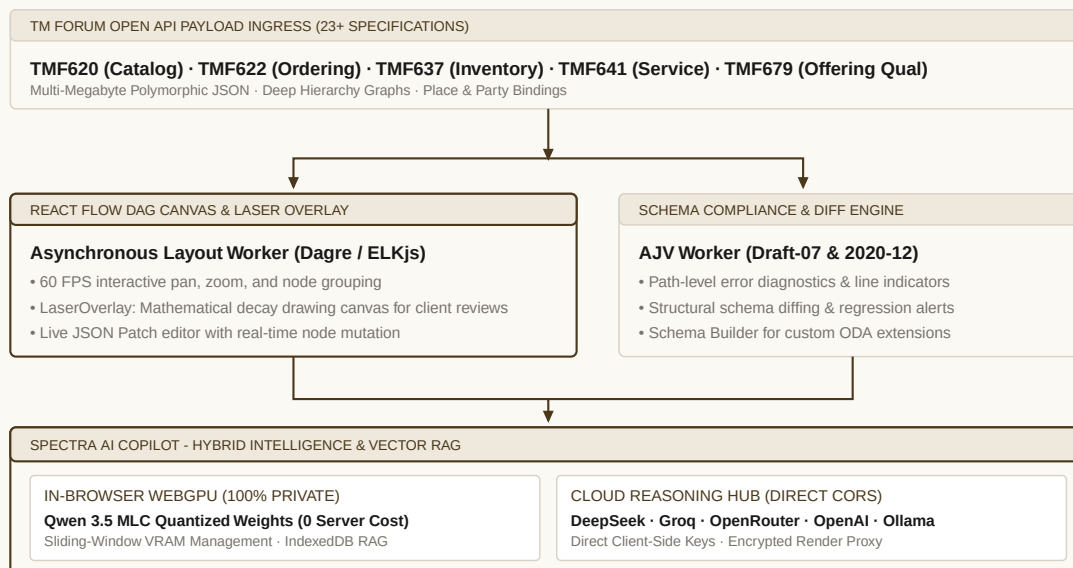
From pasted JSON to presented, validated, AI-explained topology.

Five layers, all in the browser tab. No server required for any of them.

TM FORUM API VISUALIZER & SPECTRA AI COPILOT ARCHITECTURE



Client-side single-page architecture orchestrating React Flow DAG layout workers, SVG laser drawing canvas, AJV schema compliance, and hybrid WebGPU/Cloud Spectra AI inference.



Full system schematic - TM Forum API Visualizer & Spectra AI Copilot Architecture.

Boundary 01 / 06

Payload Ingress

Parse megabytes without fear.

PURPOSE

Accept hostile real-world payloads - huge, nested, circular - and normalize them for layout.

BOUNDARY

It structures data. It renders nothing and judges nothing about correctness.

FAILURE PREVENTED

Tab crashes and frozen renderers on enterprise-scale payloads.

Boundary 02 / 06

Layout Worker

Graphs computed off the UI thread.

PURPOSE

Compute DAG layouts for hundreds of nodes without dropping animation frames.

BOUNDARY

Geometry only. It never touches the DOM or the render loop.

FAILURE PREVENTED

Jank and input lag while large topologies arrange themselves.

Boundary 03 / 06

DAG Canvas

Topology you can navigate.

PURPOSE

Turn abstract contracts into a graph architects can read at a glance.

BOUNDARY

Presentation and navigation. Validation and intelligence live downstream.

FAILURE PREVENTED

Hours lost tracing parent-child fulfillment chains through raw JSON.

Boundary 04 / 06

Laser Presentation

Draw attention, leave no mark.

PURPOSE

Let presenters guide a room's eyes across live topology without leaving the browser.

BOUNDARY

Ephemeral ink only. It annotates nothing permanently and alters no data.

FAILURE PREVENTED

Lost audiences and the context-switching of third-party annotation tools.

Boundary 05 / 06

Schema Compliance

Path-level diagnostics, live.

PURPOSE

Prove payloads against 23+ TM Forum schemas with errors you can act on.

BOUNDARY

It diagnoses and suggests patches. It never silently rewrites payloads.

FAILURE PREVENTED

Integration failures discovered in downstream systems instead of at authoring time.

Boundary 06 / 06

Spectra AI Copilot

Private inference, local or cloud.

PURPOSE

Explain payloads and suggest fixes without the data ever being obligated to leave the machine.

BOUNDARY

It advises on what you show it. Local mode guarantees nothing crosses the network at all.

FAILURE PREVENTED

Sensitive contract data transiting third-party AI services for basic comprehension.

04 · FOLLOW THE FLOW

One payload, five hops

Watch raw text become a graph you can present and question.

Now let's follow one payload from pasted text to explained insight.

01 Payload Ingress

A multi-megabyte TM Forum payload is pasted in. Circular references are detected, hierarchy is modeled, the tab survives.

02 Layout Worker

The hierarchy ships to a worker thread where Dagre or ELKjs computes positions. The UI thread stays at full frame rate throughout.

03 DAG Canvas

The positioned graph renders - navigable, zoomable, inspectable. A structural defect that hid in text is now visibly wrong.

04 Laser Presentation

In a live review, the presenter draws fading trails directly over the topology to walk stakeholders through the defect. No plugins, no screenshots.

05 Schema Compliance

Validation runs against the matching TM Forum schema and reports the exact paths that violate it, with patch suggestions.

06 Spectra AI Copilot

The architect asks what a construct means. A local WebGPU model answers from the payload and cited specs - nothing leaves the machine.

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.

React Flow & Directed Acyclic Graph (DAG) Layouts vs. Static Collapsible JSON Trees

Nested JSON trees obscure topological relationships between parent/child order items, billing accounts, and qualifying characteristics.

CHOSEN

React Flow graph canvas orchestrated with Dagre and ELKjs layout engines inside dedicated Web Workers.

WHY

Transforms abstract data contracts into an intuitive visual graph where architects can navigate dependencies, inspect polymorphic attributes, and understand fulfillment topology in real time.

ALTERNATIVES CONSIDERED

- Collapsible JSON tree renderers
- Raw code editors with syntax highlighting
- Custom SVG canvas from scratch

In-Browser WebGPU Local AI + Native CORS Cloud Hub vs. Centralized Heavy AI Server

Enterprise telco payloads contain commercially sensitive pricing, network topologies, and customer identities that cannot touch third-party servers.

CHOSEN

Hybrid AI architecture featuring in-browser WebGPU inference via @mlc-ai/web-llm (Qwen 3.5) paired with native browser CORS for cloud models.

WHY

Provides 100% data privacy with zero server compute costs. Prompts and weights run purely in local browser memory and WebGPU vRAM.

ALTERNATIVES CONSIDERED

- Centralized Node.js backend streaming all prompts to OpenAI
- Requiring every user to install local Python environments
- Static rule-based validation scripts only

Real-Time Laser Pointer Presentation Canvas vs. Screen Annotation Plugins

Architects frequently conduct live virtual sessions to explain complex payload dependencies to stakeholders without screen clutter.

CHOSEN

Custom SVG pointer drawing surface (LaserOverlay) with fading quadratic bezier trails and 1500ms glow decay.

WHY

Allows presenters to naturally guide client attention directly on active nodes without leaving the browser or installing extra software.

ALTERNATIVES CONSIDERED

- Third-party zoom annotation tools
- Static screenshot markup tools
- Mouse cursor highlighting

AJV Schema Compliance & Live JSON Patching vs. Read-Only Static Validators

Architects not only need to find errors in payloads; they need to simulate fixes and author schema extensions on the fly.

CHOSEN

Integrated Schema Hub and Schema Builder with path-level JSON patch generation and instantaneous visual feedback.

WHY

Accelerates new TMF API development cycles by allowing engineers to modify payloads and verify compliance live.

ALTERNATIVES CONSIDERED

- Standard post-call JSON linter
- Online schema validator sites
- CLI-only validation scripts

CHALLENGE THE DESIGN

The payload needs AI explanation. Where does inference physically run?

A CENTRAL BACKEND STREAMING TO A CLOUD MODEL

Simplest to build - and disqualified on arrival. Sensitive telco payloads transiting your servers to a third party fails every enterprise review that matters.

NO AI - STATIC RULES AND DOCUMENTATION

Zero privacy risk and zero comprehension help. Payloads stay as unreadable as they started, which was the original complaint.

HYBRID: LOCAL WEBGPU FIRST, DIRECT CLOUD AS FALLBACK

This is the architecture's answer. Data never has to leave the machine, capable hardware gets free private inference, and the cloud path covers the rest without your servers in the middle.

HOW THE ARCHITECT REASONED

Privacy was a placement constraint, not a feature request. Once inference had to live with the data, the hybrid design was nearly forced - and better for it.

Graph layout for hundreds of nodes blocks the UI thread for seconds. What moves?

CAP THE GRAPH SIZE

Protects the frame rate by refusing the actual workload. Enterprise payloads do not shrink because the tool prefers it.

USE A CHEAPER LAYOUT ALGORITHM

Faster and uglier - unreadable topology at full speed is still unreadable. Quality of layout is the product here.

MOVE LAYOUT TO A WEB WORKER

This is the architecture's answer. Full-quality Dagre and ELKjs layouts compute off-thread while the canvas stays interactive at sixty frames.

HOW THE ARCHITECT REASONED

The UI thread belongs to fingers. Anything measured in hundreds of milliseconds belongs in a worker - layout doubly so.

06 · FAILURE MODES

Let's try to break it

Circular payloads, schema drift, and GPU-less machines - each with a landing place.

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

The circular payload

WHAT GOES WRONG

A payload references itself - the shape that kills naive tree renderers.

WHERE IT LANDS

Ingress detects cycles during modeling and represents them explicitly. The graph renders references as links, not infinite recursion.

The machine without WebGPU

WHAT GOES WRONG

A stakeholder's laptop cannot run local models - no GPU, no weights, no patience.

WHERE IT LANDS

The copilot degrades to direct-CORS cloud models or local Ollama with the same interface. Privacy posture changes explicitly, never silently.

The payload from a newer spec

WHAT GOES WRONG

A partner sends TMF data against a schema revision the validator has never seen.

WHERE IT LANDS

Version-aware AJV packs report precisely what is unknown versus what is wrong, and the schema builder covers custom ODA extensions.

The thousand-node order

WHAT GOES WRONG

An order payload expands into a node count that would freeze main-thread layout.

WHERE IT LANDS

Layout never runs on the UI thread, so the worst case is a progress state on the canvas - never a dead tab.

The payload that must not travel

WHAT GOES WRONG

A contract payload arrives under strict instructions: analyze, but nothing leaves this machine.

WHERE IT LANDS

Local inference plus IndexedDB RAG complete the whole loop in-browser. The architecture treats this as the normal case, not a special mode.

STANDING COMPROMISES

In-Browser WebGPU Compute vs Cloud API Model Capabilities

Gain: Zero server hosting cost, zero token billing, and total compliance with strict enterprise data residency rules.

Cost: Requires client device with WebGPU support (Chrome/Edge/Brave) and initial download of quantized model weights (~500MB - 1.5GB).

Mitigation: Engineered a dual-engine Model Hub: users without WebGPU can connect lightweight cloud keys (DeepSeek/Groq) with native CORS or run local Ollama/LM Studio.

Full Client-Side Schema Validation vs Server-Side Verification

Gain: Instant sub-millisecond feedback on every keystroke without network latency.

Cost: Requires packaging schemas for 23 TM Forum APIs into client bundle chunks.

Mitigation: Employed dynamic schema registry caching and lazy-loading for API definitions (TMF620-TMF764).

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 suite is strongest where payloads are large, structured, and sensitive: exactly the workloads generic tools abandon. Visualization plus validation plus private explanation compound - each layer makes the others more valuable.

Its limits are stated, not hidden. Client hardware bounds the local AI story, schema packs must be maintained as specs evolve, and a browser tab is still a browser tab for the truly enormous.

WHAT I WOULD CHANGE TODAY

- Stream layout progressively so thousand-node graphs become navigable before finishing
- Version schema packs as data updates independent of application deploys
- Benchmark local model quality per TM Forum domain and publish the gaps

VERIFIED OUTCOMES

- Deployed live to production at tmf.victorkuldeep.com and Cloudflare Workers UAT mirror with global sub-second availability
- Comprehensive visual, structural, and compliance support for 23 TM Forum Open APIs spanning Product, Service, Resource, Customer, and Billing
- Zero server token costs: 100% private in-browser WebGPU AI inference with Qwen 3.5 local models
- Adopted by architects and BSS engineers for interactive client design sessions using the built-in laser pointer presentation tool
- Accelerated schema compliance and payload debugging from hours of manual JSON parsing to instant visual diagnosis

Continue the exploration

[Interactive exhibit: victorkuldeep.com/case-studies/tmf-api-visualizer-spectra-ai/deep-dive](https://victorkuldeep.com/case-studies/tmf-api-visualizer-spectra-ai/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



KULDEEP SINGH



CASE STUDY 05 · DEEP DIVE EXHIBIT · ~6 MIN

BrewCMS: Open-Source Content Control Plane & Agent Operating System

“When traditional CMS architectures force false choices between fragile headless SaaS subscription bills and bloated legacy monolithic platforms, architect content as a governed, database-agnostic control plane with bounded agent autonomy and zero SaaS lock-in.”

ROLE

Creator & Principal Systems Architect

TIMELINE

Open Source Systems Initiative · 2026

DOMAIN

Open Source & Content Systems

KULDEEP SINGH · DIGITAL ARCHITECTURE

What was built, and why

Modern development teams face an escalating content infrastructure crisis: traditional headless CMS platforms (Contentful, Sanity, Strapi Cloud) charge escalating SaaS subscriptions while introducing 20ms–80ms HTTP network latency and vendor lock-in. Meanwhile, AI coding agents (Claude, Cursor, Gemini) are increasingly asked to manipulate content but lack safe, bounded governance mechanisms. To solve this, created BrewCMS: an open-source, database-agnostic Content Control Plane for modern Next.js applications. Built around a pure domain core decoupled from any framework or database driver, it compiles Markdown and Frontmatter into cryptographic Content Intermediate Representations (IR) anchored by SHA-256 hashes. Storage is governed via pluggable storage adapters: running in-process via Node.js 22 native node:sqlite with WAL mode for sub-0.2ms zero-daemon latency, or scaling out to PostgreSQL and MySQL for distributed clusters. BrewCMS operates in two flexible topologies: embedded in-process directly inside Next.js App Router applications (consuming zero extra hosting webapp slots and zero subdomains) or as a standalone headless control plane. It features a native Model Context Protocol (MCP) server providing AI agents with bounded, audited content tools (brew_list_documents, brew_create_draft, brew_request_publish) subject to explicit human editorial policy gates. Proven in production as the content engine for CoffeeDiscussions and this developer portfolio.

As digital platforms scale, content management systems have bifurcated into two undesirable extremes: monolithic legacy platforms with excessive memory and plugin overhead, or API-first SaaS headless platforms that separate content from presentation at the cost of high monthly bills, external network latency, and brittle third-party dependencies. Furthermore, with the rise of autonomous AI coding assistants (Cursor, Claude Desktop, Antigravity), engineering teams need AI agents to inspect, draft, and propose content modifications directly. However, giving AI agents unconstrained write access to databases creates severe risks of hallucinations and brand drift. A new architecture was needed: a local-first, zero-dependency Content Operating System that combines sub-millisecond in-process performance with formal, human-gated AI autonomy.

1) Headless CMS SaaS platforms introduce recurring subscription costs, vendor lock-in, and 20ms–80ms HTTP latency round-trips for every document query. 2) Traditional Node.js SQLite solutions depend on better-sqlite3 or native C++ addons that fail on modern serverless or lightweight hosting environments. 3) Content revisions in standard CMS databases are mutable, causing unversioned content drift without cryptographic audit trails. 4) AI agents lack a standard, governed interface to read and write content safely without bypassing editorial review.

CONSTRAINTS

- Zero native C++ build dependencies: Must run on Node.js 22 native node:sqlite without node-gyp.
- Sub-millisecond read latency: Public document queries must execute in <0.2ms directly from disk B-trees.
- Bounded AI agent autonomy: Autonomous operations must be subject to policy gates (ALLOW, DENY, REQUIRE_APPROVAL).

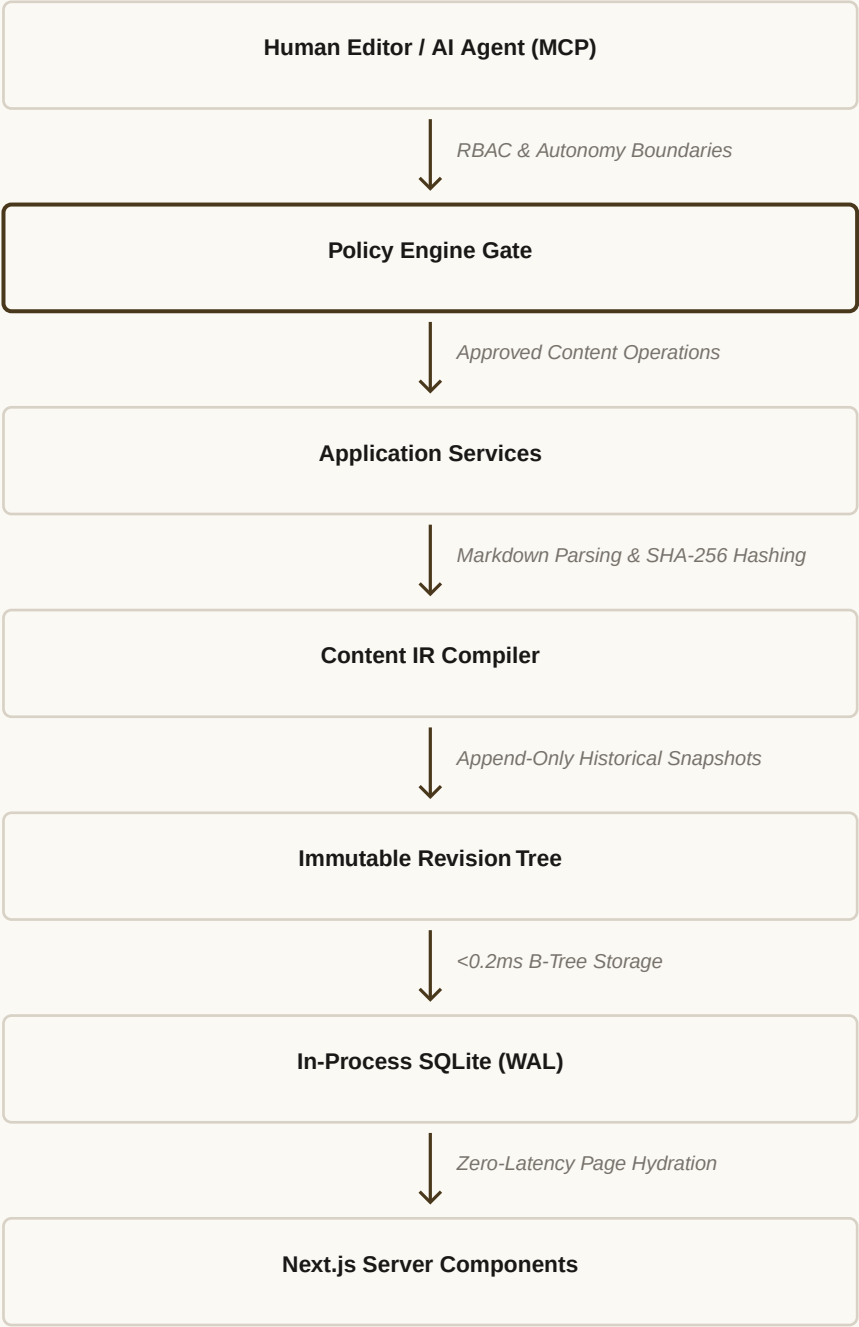
- Dual-topology flexibility: Must run embedded inside existing Next.js apps or standalone as a headless REST server.
- Zero SaaS subscription fees: 100% self-hosted on commodity disk without external database infrastructure.

PART I · ARCHITECTURE

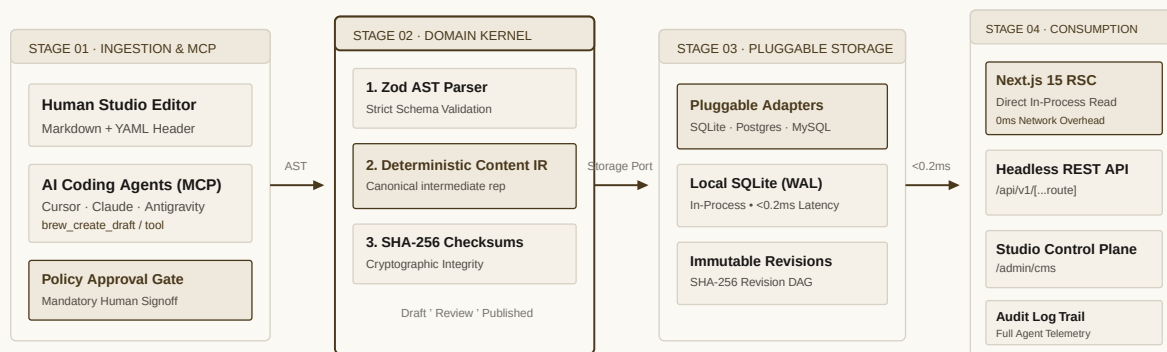
The system as designed

BrewCMS implements a strict Hexagonal / Ports & Adapters architecture. At the center is `@brew-cms/core`, containing pure TypeScript domain entities, business logic, and application services with zero framework dependencies. The content layer (`@brew-cms/content`) tokenizes Markdown and Frontmatter into a versioned Abstract Syntax Tree / Content IR, computing a deterministic SHA-256 hash for every revision. Storage (`@brew-cms/db`) is abstracted via repository interfaces implemented on top of Node.js 22 native `node:sqlite` with Write-Ahead Logging (WAL) and 5000ms busy timeouts for high-concurrency read/write operations. AI agents interact through a native Model Context Protocol server (`@brew-cms/mcp`) that maps agent tool calls (`brew_create_draft`, `brew_request_publish`) to core application services, evaluating every mutation against a declarative policy engine (`@brew-cms/policy`) before execution. In consumer applications, BrewCMS embeds directly inside the Next.js App Router, enabling Server Components to query documents in-process with zero network overhead while exposing the full Editorial Studio at `/admin/cms`.

BREWCMS GOVERNED CONTENT OPERATING SYSTEM & AGENT CONTROL PLANE



Hexagonal pipeline showing human editorial studio, bounded AI agents via MCP, deterministic Content IR compiler, and in-process SQLite storage.



Full system schematic - BrewCMS Governed Content Operating System & Agent Control Plane.

PART I · DECISIONS

Choices, engineering, tradeoffs

Native node:sqlite with WAL Mode over external database or native addons

Traditional embedded SQLite in Node.js required better-sqlite3, which requires native C++ compilation tools (python, make, gcc) that frequently break in containerized or shared hosting environments.

CHOSEN

Node.js 22 built-in node:sqlite with PRAGMA journal_mode = WAL

WHY

Zero npm build scripts, zero native C++ compilation requirements, ~15MB LRU memory cache footprint, and sub-0.2ms read latency directly on NVMe disk. Fits within any standard Node hosting quota without managing a separate database daemon.

Deterministic Content IR with SHA-256 Hashing over raw Markdown storage

Storing raw markdown strings causes parsing ambiguities, unverified HTML injection risks, and makes detecting exact semantic content modifications difficult across revision histories.

CHOSEN

Deterministic Content IR (Abstract Syntax Tree) validated by Zod schemas with cryptographic content hashes

WHY

Decouples storage from rendering, allows safe client and server rendering without dangerouslySetInnerHTML hazards, and computes exact cryptographic content hashes for immutable revision tracking.

Embedded In-Process Engine over Mandatory Remote Headless Server

Deploying a separate CMS server requires dedicated ports, subdomain registration (cms.domain.com), cross-origin CORS handling, and extra webapp slots on hosting platforms like Hostinger.

CHOSEN

Dual-mode architecture supporting both embedded in-process direct calls and standalone REST API

WHY

Allows Next.js applications like CoffeeDiscussions and this portfolio to run the entire CMS natively inside /admin/cms using 0 extra webapp slots, 0 subdomains, and zero HTTP network round-trips.

Native Model Context Protocol (MCP) with Human Approval Policy Gates

AI agents need structured access to query documents, propose drafts, and update taxonomies without being given unfettered write or publish permissions.

CHOSEN

Standardized Model Context Protocol (MCP) server with @brew-cms/policy enforcement

WHY

Provides universal compatibility with Cursor, Claude Desktop, and autonomous agent loops. Low-risk operations (reading, drafting) are allowed automatically, while high-risk operations (publishing, unpublishing) generate approval requests for human editors.

ENGINEERING NOTES

- 13 Modular TypeScript Packages: Built as a strict pnpm monorepo using TypeScript project references (tsc -b), ensuring clean dependency graphs and zero circular imports.
- High-Concurrency WAL Concurrency: SQLite configured with PRAGMA journal_mode = WAL, synchronous = NORMAL, and busy_timeout = 5000, allowing concurrent readers alongside write operations without database locking errors.
- Full Next.js 15 App Router Editorial Studio: Responsive studio featuring split-pane live markdown editing, real-time HTML preview, revision rollback history, media library asset browser, taxonomy governance, and agent approval queues.
- Zero-Latency Server Component Hydration: In embedded mode, Next.js Server Components call repository methods directly in-process, bypassing the HTTP network stack entirely.
- Cryptographic Audit Trail: Every state transition, human editorial decision, and AI agent execution emits an immutable audit event recording actor identity, before/after snapshots, and timestamps.

PART I · OUTCOMES

Tradeoffs, results, lessons

Database Storage

Gain: Zero external server cost, zero connection latency, and instant local-first execution via node:sqlite.

Cost: Multiple applications maintain independent SQLite database files on disk rather than a single unified database.

Mitigation: Supported through BrewCMS REST API export/import pipelines and optional standalone headless deployment topology.

AI Agent Autonomy

Gain: Guaranteed editorial integrity and zero unauthorized content publishing or hallucinations.

Cost: AI agents cannot publish documents to live production without a human editor approving the action run.

Mitigation: Integrated one-click human approval queue in the Studio UI and automated notification hooks.

OUTCOMES

- 100% elimination of third-party headless CMS SaaS subscription costs across all reference applications.
- Reduced content retrieval latency from ~30ms HTTP round-trips down to sub-0.2ms in-process SQLite B-tree queries.
- Consumes only 2 out of 5 allowed webapp slots on Hostinger, leaving 3 slots completely free for future applications.
- Full test suite passing: 12 test suites, 45/45 tests passing with 100% clean typecheck (0 errors).
- Full production adoption across CoffeeDiscussions (magazine) and VictorKuldeep developer portfolio.

LESSONS

- Local-first SQLite with WAL mode is vastly superior to remote headless APIs for single-server and shared-hosting Next.js architectures.
- AI agents in production need formal policy boundaries and approval queues rather than direct database write permissions.
- Deterministic Content IR with cryptographic hashing eliminates content drift and makes revision rollbacks completely predictable.

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 walk through a content system the way its data flows - not the way its marketing reads.

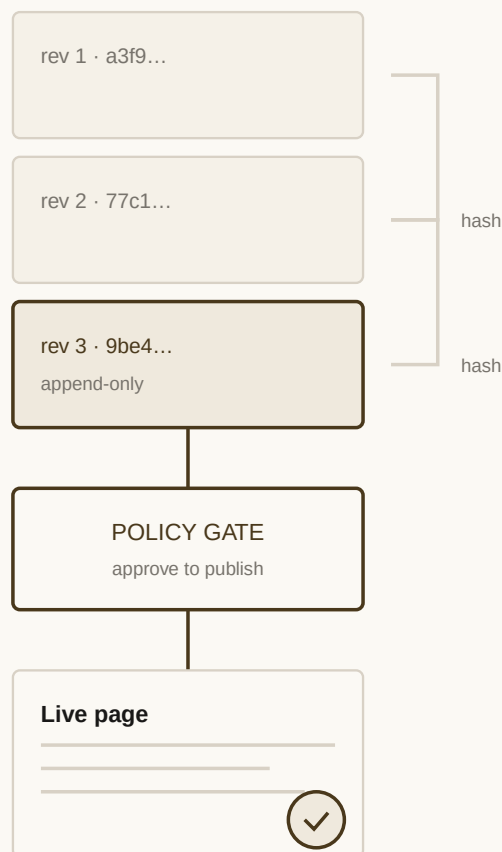


Exhibit plate - the system at a glance.

This exhibit traces BrewCMS, an open-source content control plane: Markdown enters, a deterministic compiler turns it into hashed revisions, policy gates guard publishing, and AI agents operate through bounded tools instead of raw database access.

Everything here is public open-source work. The repositories, the revision model, and the tradeoffs are all inspectable outside this page - the exhibit just orders them the way an architect would explain them.

How to use this exhibit

- Inspect each boundary to see what it guarantees
- Step one document from draft to published page
- Decide where you would let an AI agent touch content
- Break the publishing flow, then watch the gates catch it

02 · THE PROBLEM

Two bad options

Subscription lock-in on one side, monolith drag on the other - plus agents with no safe door.

Teams do not choose bad content infrastructure. They inherit two bad options.

Headless SaaS platforms charge a growing subscription, add tens of milliseconds per query, and hold an exit tax over your content. Legacy monoliths cost nothing monthly and everything in agility. Most teams pick one and pay the other price quietly.

Meanwhile AI coding assistants are asked to edit content with no safe interface - either they cannot touch it, or they can touch all of it. Both answers are wrong, and the second one is dangerous.

THE CONSTRAINTS THIS FAILURE IMPOSES

- Zero native C++ build dependencies: Must run on Node.js 22 native node:sqlite without node-gyp.
- Sub-millisecond read latency: Public document queries must execute in <0.2ms directly from disk B-trees.
- Bounded AI agent autonomy: Autonomous operations must be subject to policy gates (ALLOW, DENY, REQUIRE_APPROVAL).
- Dual-topology flexibility: Must run embedded inside existing Next.js apps or standalone as a headless REST server.
- Zero SaaS subscription fees: 100% self-hosted on commodity disk without external database infrastructure.

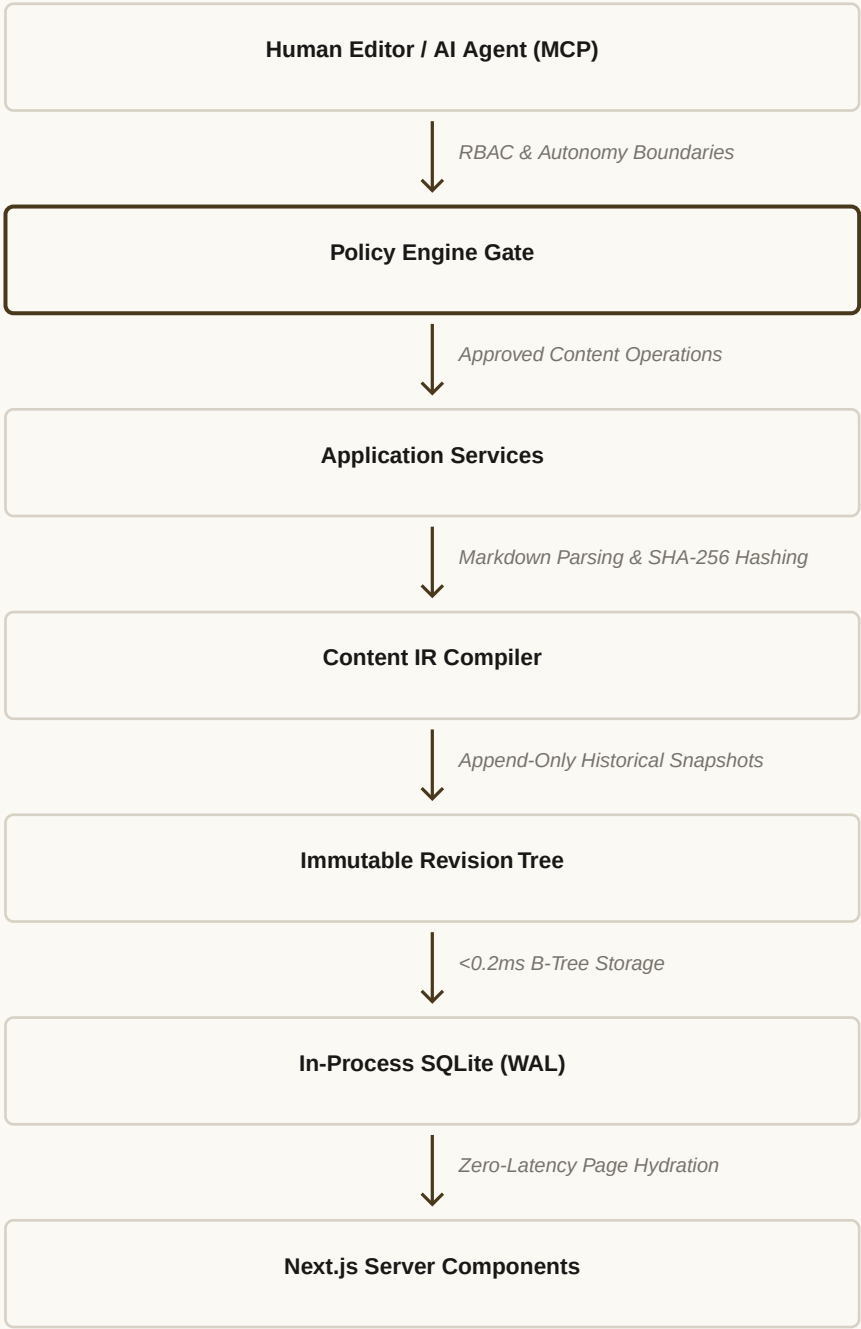
03 · THE ARCHITECTURE

Inspect the system

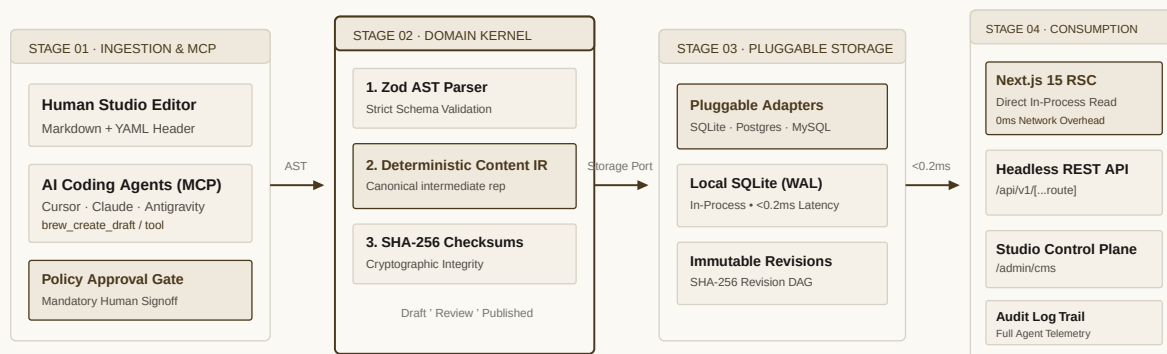
Follow content from keystroke to published page through six governed boundaries.

Six boundaries. Each one removes a class of surprise.

BREWCMS GOVERNED CONTENT OPERATING SYSTEM & AGENT CONTROL PLANE



Hexagonal pipeline showing human editorial studio, bounded AI agents via MCP, deterministic Content IR compiler, and in-process SQLite storage.



Full system schematic - BrewCMS Governed Content Operating System & Agent Control Plane.

Boundary 01 / 06

Authoring Surface

Write Markdown, nothing else.

PURPOSE

Give humans the simplest possible input contract: Markdown with frontmatter.

BOUNDARY

Content in, never layout out. Rendering is someone else's job downstream.

FAILURE PREVENTED

Theme lock-in, where a redesign means rewriting every document.

Boundary 02 / 06

Content IR Compiler

Deterministic parse to typed IR.

PURPOSE

Turn ambiguous Markdown into a typed intermediate representation, identically every time.

BOUNDARY

Pure function of the input text. Same bytes in, same tree and hash out - no clock, no randomness.

FAILURE PREVENTED

Silent content drift and unverified HTML slipping into stored documents.

Boundary 03 / 06

Immutable Revision Tree

Append-only history by hash.

PURPOSE

Keep every version of every document as an immutable, content-addressed snapshot.

BOUNDARY

History is append-only. Rollback means pointing at an older hash, never rewriting.

FAILURE PREVENTED

Unversioned overwrites and the 'who changed this and when' investigations that follow.

Boundary 04 / 06

Policy Gate

Allow, deny, or require approval.

PURPOSE

Decide what every write - human or agent - is allowed to do before it happens.

BOUNDARY

It judges intent, never content quality. Taste stays human; permission stays formal.

FAILURE PREVENTED

Unauthorized publishes and agent hallucinations reaching readers.

Boundary 05 / 06

Pluggable Storage

SQLite local, Postgres at scale.

PURPOSE

Persist revisions behind repository interfaces the core never sees past.

BOUNDARY

The domain core holds zero database drivers. Adapters are swappable without touching logic.

FAILURE PREVENTED

Infrastructure lock-in, where growth means replatforming instead of reconfiguring.

Boundary 06 / 06

MCP Agent Server

Bounded tools, audited calls.

PURPOSE

Let AI agents work with content through the same gates humans pass.

BOUNDARY

Agents propose and draft. Publishing across the gate still needs a human where policy says so.

FAILURE PREVENTED

Unconstrained agent writes - brand drift, hallucinations, and silent overwrites.

04 · FOLLOW THE FLOW

One document, six hops

Watch a draft earn its way to the page - nothing skips the gates.

Now let's follow one document from keystroke to published page.

01 Authoring Surface

An editor writes Markdown with frontmatter. No layout decisions are possible here, which is what keeps the content portable.

02 Content IR Compiler

The compiler parses and validates the text into a typed tree and hashes it. Identical bytes always produce the identical hash.

03 Immutable Revision Tree

The new tree appends as an immutable revision. The past is untouched; rollback is just pointing at an older hash.

04 Policy Gate

The publish request meets policy. Low-risk drafts pass; publishing requires the approval the policy names.

05 Pluggable Storage

The approved revision persists through the configured adapter - local SQLite or Postgres, the core cannot tell.

06 MCP Agent Server

Later, an AI agent drafts a follow-up through bounded MCP tools. Its draft enters the same pipeline at the same gate - no side doors.

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.

Native node:sqlite with WAL Mode over external database or native addons

Traditional embedded SQLite in Node.js required better-sqlite3, which requires native C++ compilation tools (python, make, gcc) that frequently break in containerized or shared hosting environments.

CHOSEN

Node.js 22 built-in node:sqlite with PRAGMA journal_mode = WAL

WHY

Zero npm build scripts, zero native C++ compilation requirements, ~15MB LRU memory cache footprint, and sub-0.2ms read latency directly on NVMe disk. Fits within any standard Node hosting quota without managing a separate database daemon.

ALTERNATIVES CONSIDERED

- PostgreSQL
- better-sqlite3
- Supabase
- Cloudflare D1

Deterministic Content IR with SHA-256 Hashing over raw Markdown storage

Storing raw markdown strings causes parsing ambiguities, unverified HTML injection risks, and makes detecting exact semantic content modifications difficult across revision histories.

CHOSEN

Deterministic Content IR (Abstract Syntax Tree) validated by Zod schemas with cryptographic content hashes

WHY

Decouples storage from rendering, allows safe client and server rendering without dangerouslySetInnerHTML hazards, and computes exact cryptographic content hashes for immutable revision tracking.

ALTERNATIVES CONSIDERED

- Raw markdown text
- HTML string blobs
- ProseMirror JSON

Embedded In-Process Engine over Mandatory Remote Headless Server

Deploying a separate CMS server requires dedicated ports, subdomain registration (cms.domain.com), cross-origin CORS handling, and extra webapp slots on hosting platforms like Hostinger.

CHOSEN

Dual-mode architecture supporting both embedded in-process direct calls and standalone REST API

WHY

Allows Next.js applications like CoffeeDiscussions and this portfolio to run the entire CMS natively inside /admin/cms using 0 extra webapp slots, 0 subdomains, and zero HTTP network round-trips.

ALTERNATIVES CONSIDERED

- Mandatory Headless API
- Microservices container
- Third-party SaaS

Native Model Context Protocol (MCP) with Human Approval Policy Gates

AI agents need structured access to query documents, propose drafts, and update taxonomies without being given unfettered write or publish permissions.

CHOSEN

Standardized Model Context Protocol (MCP) server with @brew-cms/policy enforcement

WHY

Provides universal compatibility with Cursor, Claude Desktop, and autonomous agent loops. Low-risk operations (reading, drafting) are allowed automatically, while high-risk operations (publishing, unpublishing) generate approval requests for human editors.

ALTERNATIVES CONSIDERED

- Custom OpenAI function calling
- Unrestricted REST API tokens
- LangChain agent tools

CHALLENGE THE DESIGN

An AI agent needs to fix a typo across forty documents. How much access does it get?

DIRECT WRITE ACCESS - IT IS JUST TYPOS

Fast until the forty-first document, where 'just typos' meets a hallucinated paragraph. Blast radius is the whole corpus with no record of intent.

NO ACCESS - AGENTS STAY READ-ONLY

Safe and increasingly useless. Every agentic workflow becomes manual labor, and teams route around you with copy-paste.

BOUNDED DRAFT TOOLS BEHIND POLICY GATES

This is the architecture's answer. The agent drafts at machine speed, policy decides what ships, and the revision tree records all of it.

HOW THE ARCHITECT REASONED

Drafting is low-risk and automatable; publishing is high-risk and gated. Splitting those two verbs is what makes agents safe to invite in.

The project starts on a single small server but might grow to a cluster. Which storage?

POSTGRES FROM DAY ONE

Correct for the cluster you do not have yet, and a server to operate for the site you actually have. Most projects never outgrow this tax.

SQLITE FOREVER, NO ABSTRACTION

Lovely until growth arrives, at which point every query is coupled to a file and migration means surgery.

REPOSITORY INTERFACES, SQLITE DEFAULT

This is the architecture's answer. Sub-millisecond local reads today, Postgres tomorrow, and the domain core never learns which one it is.

HOW THE ARCHITECT REASONED

The abstraction exists so the deployment can change shape without the application noticing. Small stays cheap, large stays possible.

06 · FAILURE MODES

Let's try to break it

Drift, rogue agents, and storage surprises - each with a landing place.

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

The silently edited document

WHAT GOES WRONG

A document changes without anyone recording what changed or why.

WHERE IT LANDS

Content hashes make drift mathematically visible, and the revision tree shows the exact before and after.

The agent that publishes at midnight

WHAT GOES WRONG

An autonomous loop decides a draft is good enough and ships it.

WHERE IT LANDS

Publishing crosses the policy gate, which demands the named human approval. The attempt is logged, not executed.

The site that outgrows its server

WHAT GOES WRONG

Traffic and content volume exceed what a single SQLite file should serve.

WHERE IT LANDS

The storage adapter swaps to Postgres behind unchanged repository interfaces. No application logic moves.

The Markdown with a surprise inside

WHAT GOES WRONG

Authored content carries embedded HTML or script payloads toward readers.

WHERE IT LANDS

Schema validation at compile time rejects or neutralizes what the IR cannot type. Nothing untyped reaches storage.

The rollback that must work

WHAT GOES WRONG

A bad publish needs undoing under pressure, with readers watching.

WHERE IT LANDS

Rollback repoints to an older content hash. The bad revision stays in history as evidence, not as the live page.

STANDING COMPROMISES

Database Storage

Gain: Zero external server cost, zero connection latency, and instant local-first execution via node:sqlite.

Cost: Multiple applications maintain independent SQLite database files on disk rather than a single unified database.

Mitigation: Supported through BrewCMS REST API export/import pipelines and optional standalone headless deployment topology.

AI Agent Autonomy

Gain: Guaranteed editorial integrity and zero unauthorized content publishing or hallucinations.

Cost: AI agents cannot publish documents to live production without a human editor approving the action run.

Mitigation: Integrated one-click human approval queue in the Studio UI and automated notification hooks.

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 architecture is strongest where content changes through review: publications, documentation, portfolios. Deterministic revisions and policy gates shine exactly where the cost of a mistake exceeds the cost of a queue.

Its limits are stated, not hidden. Real-time collaborative editing, binary asset pipelines, and multi-tenant hosting economics are outside what this exhibit claims. Each is a named next problem, not a surprise.

WHAT I WOULD CHANGE TODAY

- Prototype collaborative editing as branches over the revision tree rather than locks
- Measure the policy queue under real editorial load and tune approval granularity
- Extend MCP tools toward structured content operations beyond draft and publish

VERIFIED OUTCOMES

- 100% elimination of third-party headless CMS SaaS subscription costs across all reference applications.
- Reduced content retrieval latency from ~30ms HTTP round-trips down to sub-0.2ms in-process SQLite B-tree queries.
- Consumes only 2 out of 5 allowed webapp slots on Hostinger, leaving 3 slots completely free for future applications.
- Full test suite passing: 12 test suites, 45/45 tests passing with 100% clean typecheck (0 errors).
- Full production adoption across CoffeeDiscussions (magazine) and VictorKuldeep developer portfolio.

Continue the exploration

[Interactive exhibit: victorkuldeep.com/case-studies/brew-cms/deep-dive](https://victorkuldeep.com/case-studies/brew-cms/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



KULDEEP SINGH



CASE STUDY 06 · 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

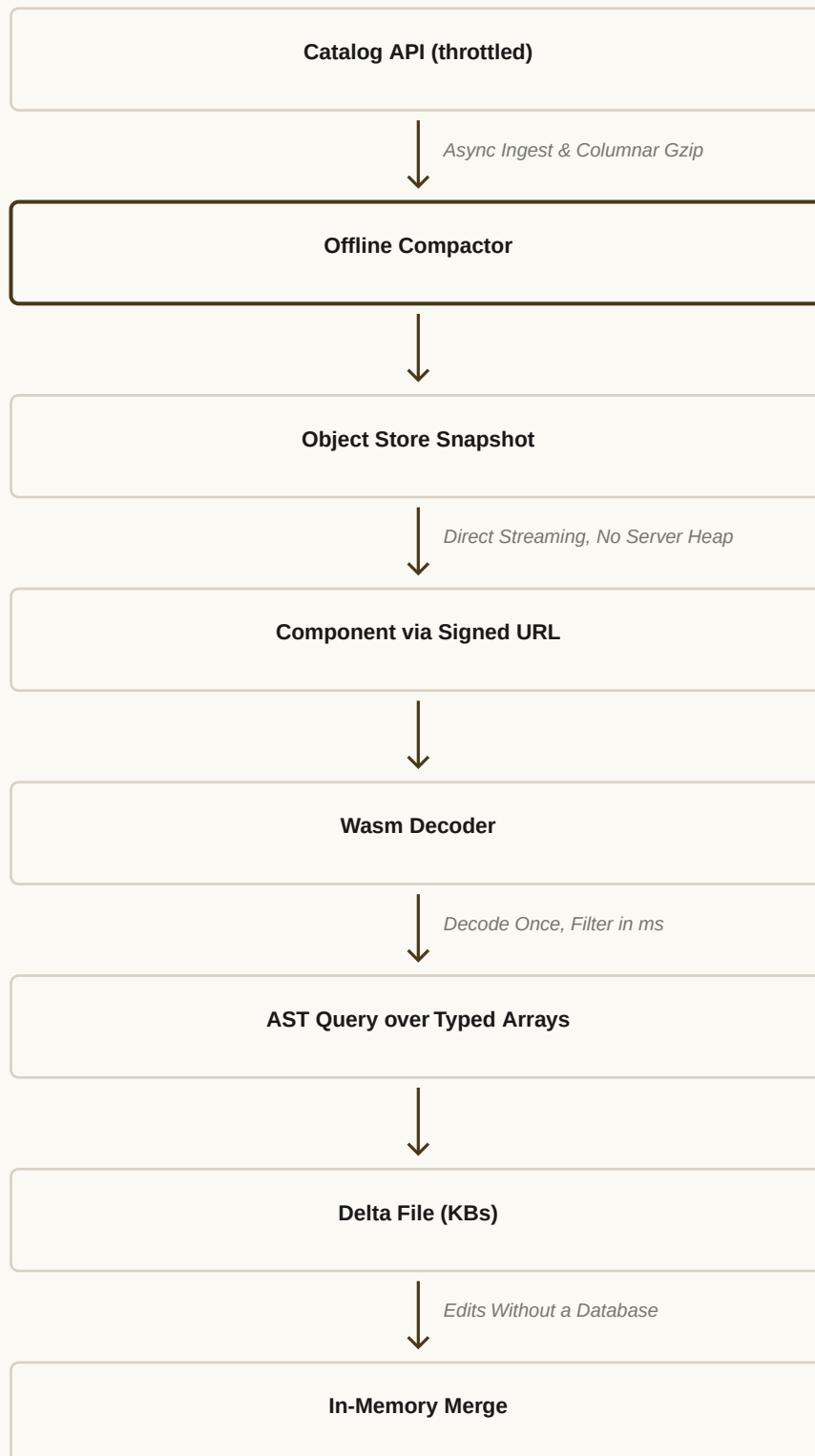
The system as designed

The pipeline I built and measured:

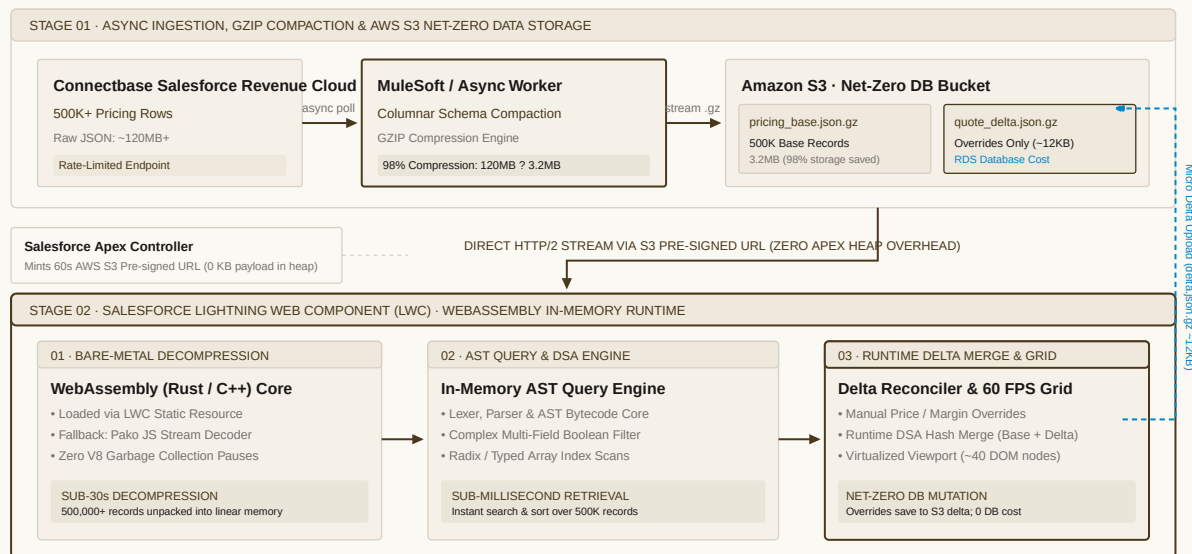
1. Offline compaction: a small worker reshapes synthetic catalog JSON into columnar arrays and gzip-compresses it. On my fixtures this lands around 2–3% of the original byte size, staged immutably on S3-compatible storage.
2. Signed-URL streaming: the server side mints a short-lived signed URL and steps 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.

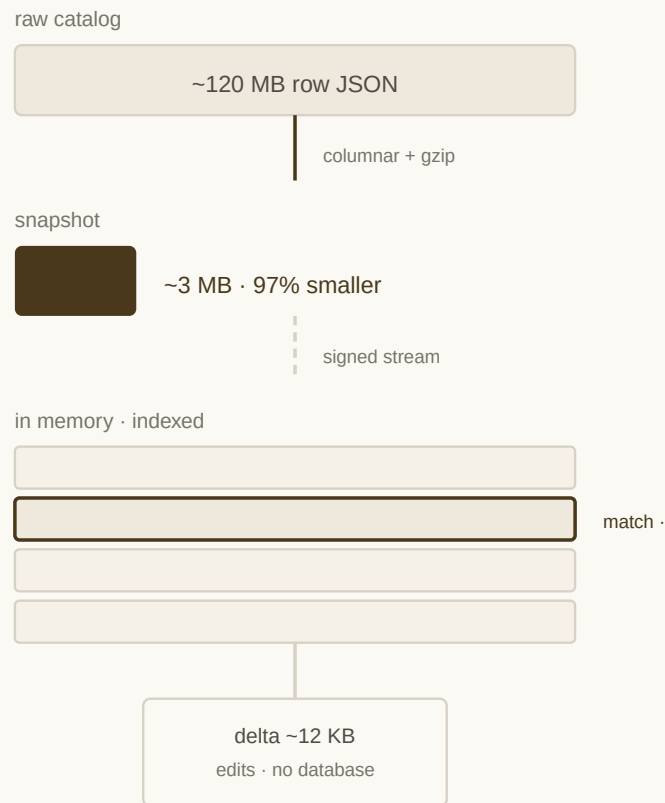


Exhibit plate - the system at a glance.

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

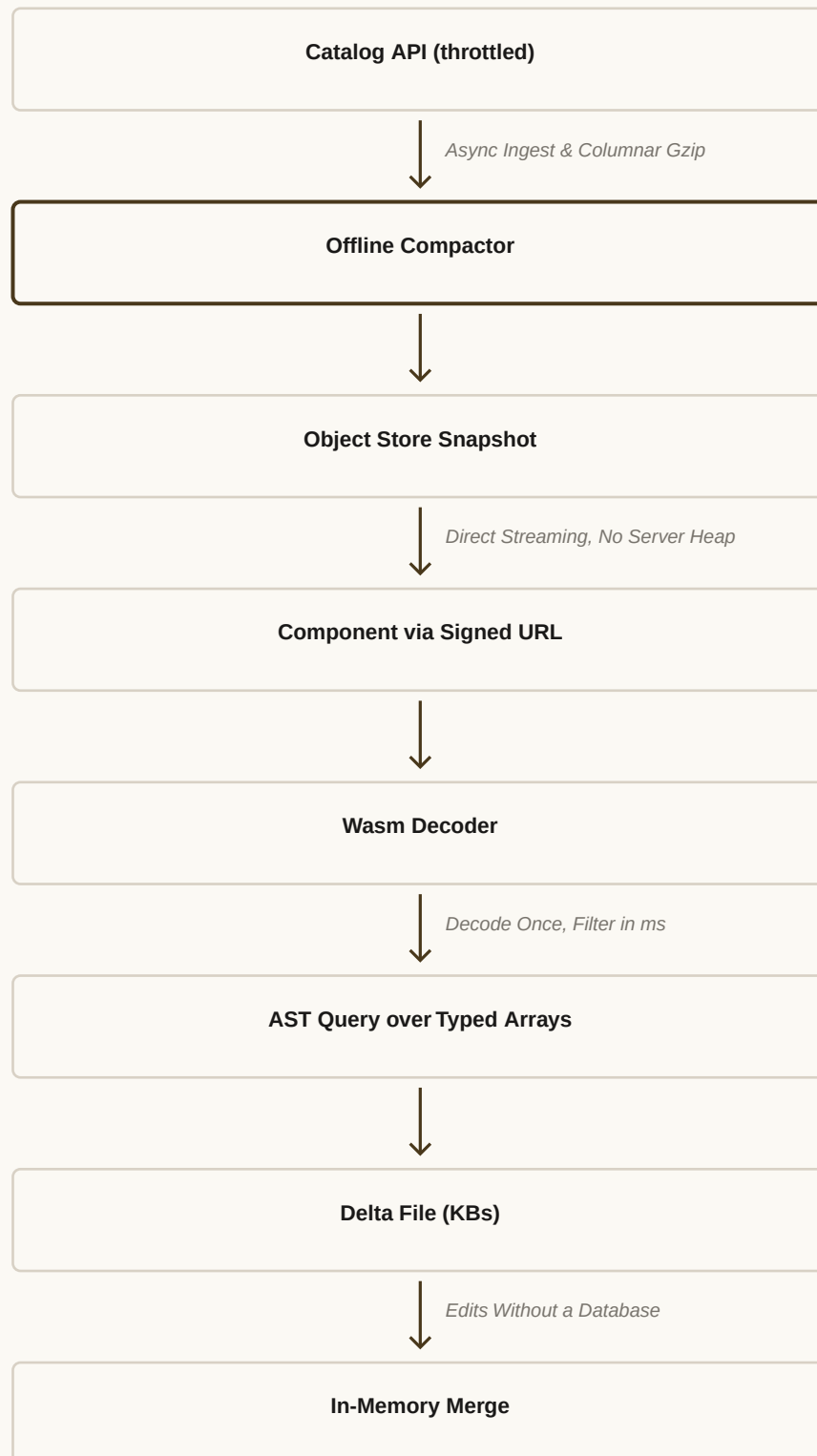
03 · THE ARCHITECTURE

Inspect the system

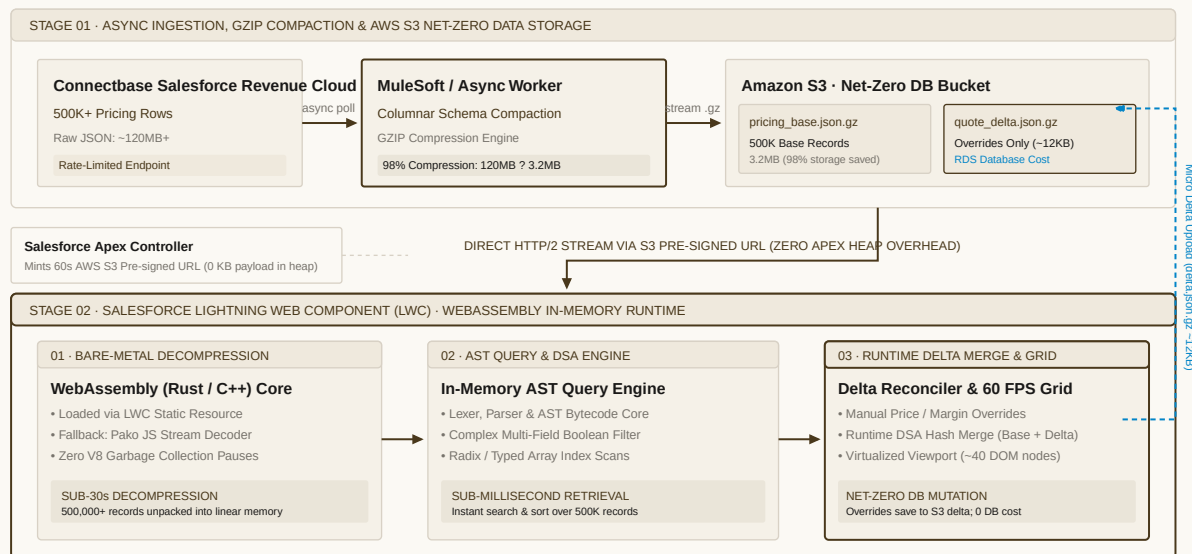
From frozen snapshot to searchable grid without touching a server heap.

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

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





CASE STUDY 07 · 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

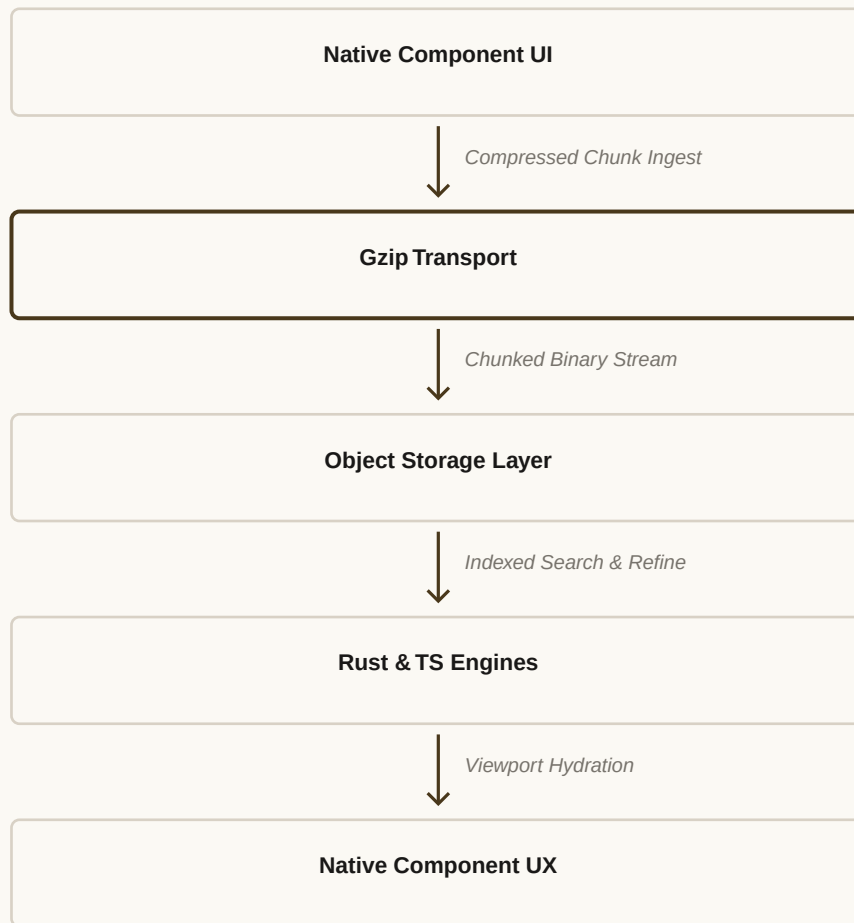
- Quoting stays inside the CRM component - no portal redirect
- No tab crashes or multi-second freezes on large fixtures
- Respect sandbox transport limits and security boundaries
- Search across locations and attributes must feel instant

PART I · ARCHITECTURE

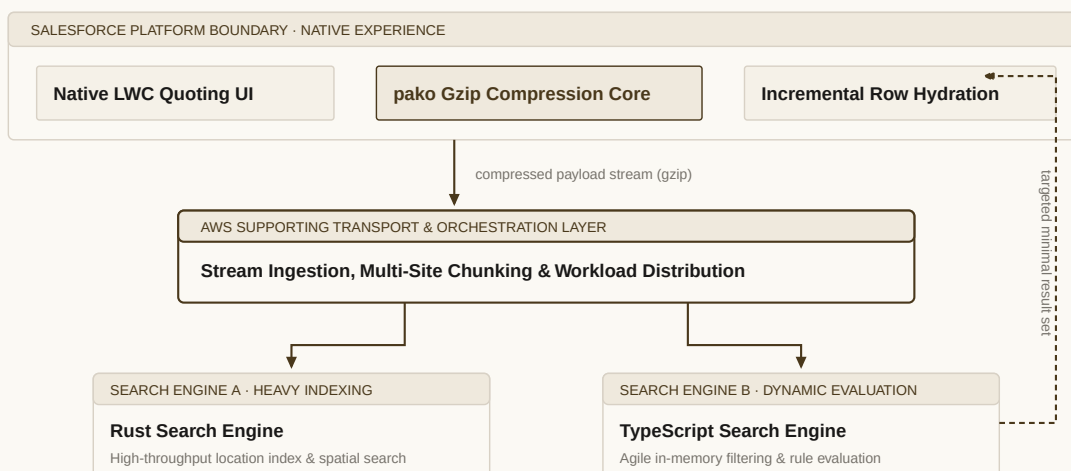
The system as designed

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.

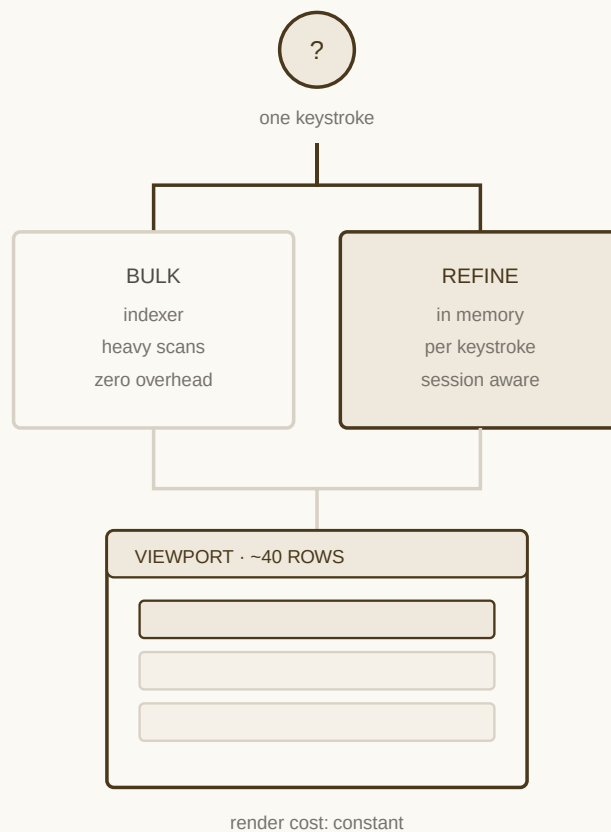


Exhibit plate - the system at a glance.

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

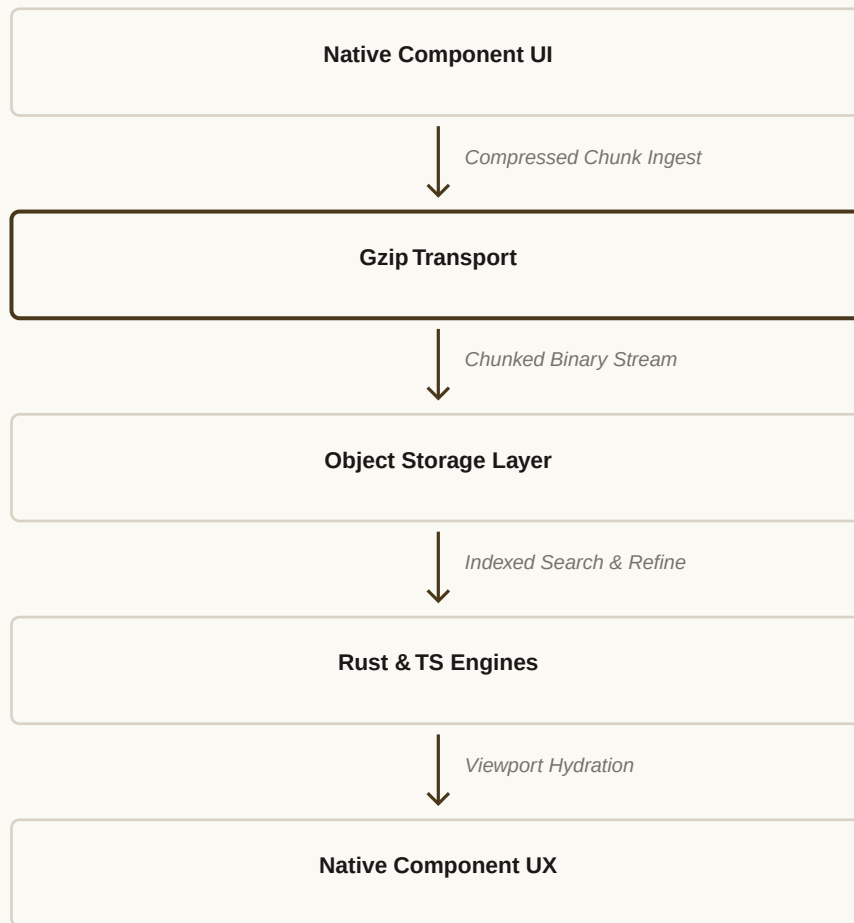
03 · THE ARCHITECTURE

Inspect the system

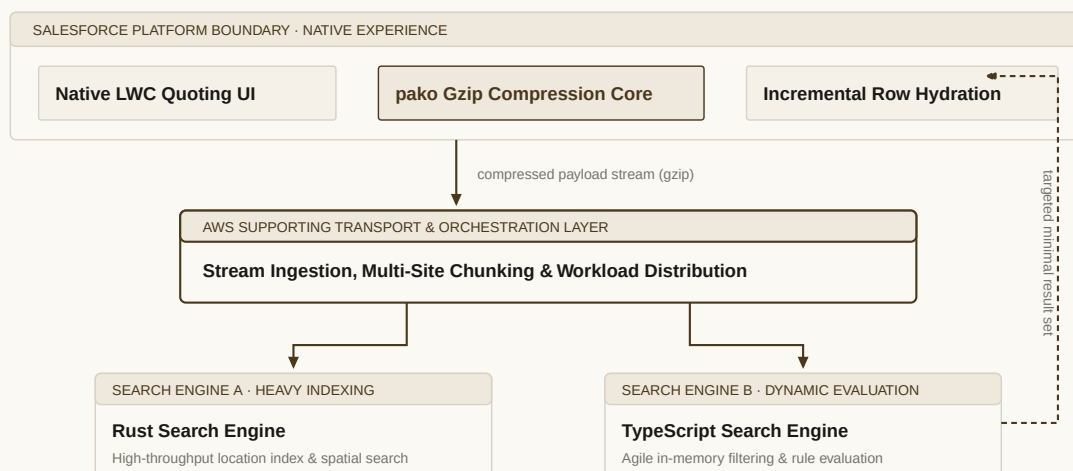
From keystroke to visible rows through compression, two engines, and a viewport.

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

Continue the exploration

[Interactive exhibit: victorkuldeep.com/case-studies/connectbase-high-volume-quoting/deep-dive](https://victorkuldeep.com/case-studies/connectbase-high-volume-quoting/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





CASE STUDY 08 · ARCHITECTURAL CASE STUDY

Save Pipelines Drawn as Graphs, Not Trigger Chains

“Make the save order visible and the failure localized - implicit trigger chains do neither.”

ROLE

Independent Architect

TIMELINE

Independent research · 2025–2026

DOMAIN

Salesforce Revenue Cloud (RCA) & Workflow Architecture

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Independent research with fixture models I defined - no employer codebase, no client objects. Multi-step saves (normalize, calculate lines, resolve tax, persist) become nodes in a declared DAG; the engine sorts them, runs independent branches where safe, and reports the exact node that failed with its inputs. I built it because I was tired of debugging save order by reading five trigger files in the wrong sequence.

Independent work, fixture models. Quoting-style saves couple calculation with persistence across several related entities, and the stages genuinely depend on each other. Implicit execution order turns every new rule into a regression risk.

Chained handlers and flag-guarded triggers hide the order, allow cycles by accident, and spread one save across many partial writes - which then fail partially and reconcile manually.

CONSTRAINTS

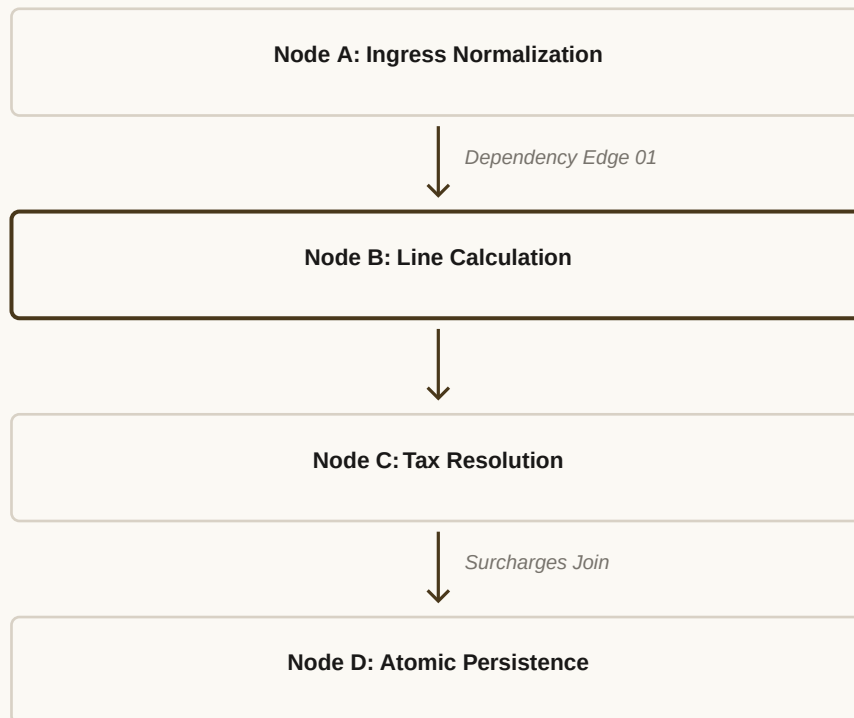
- Stages run in dependency order, stated explicitly in code
- Upstream outputs feed downstream inputs without shared mutation
- Any stage failure names the stage and preserves diagnostics
- Adding a rule means adding a node, not editing five files

PART I · ARCHITECTURE

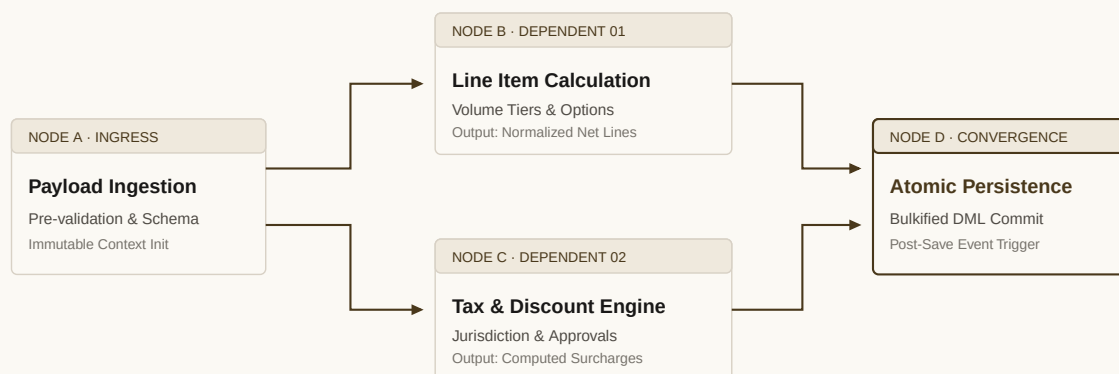
The system as designed

Each stage is a node with declared inputs, outputs, and dependencies. The engine topologically sorts the graph at startup (cycles fail fast with the offending edge named), threads an immutable context through execution, and defers persistence to a single bulk write at the end. Independent branches can run without waiting on each other.

DAG SAVE PIPELINE (STUDY MODEL)



Ingress normalizes once, calculations fan out, persistence commits once - in sorted order.



Full system schematic - DAG Save Pipeline (Study Model).

PART I · DECISIONS

Choices, engineering, tradeoffs

Declared DAG vs. chained handlers

Implicit order is the leading cause of save bugs I have debugged.

CHOSEN

Explicit graph with declared dependencies.

WHY

Order becomes reviewable. Cycles become compile-time errors. Orchestration stays separate from business rules.

Immutable context vs. shared mutable records

Shared mutation across stages produces action-at-a-distance bugs.

CHOSEN

Immutable context with explicit stage I/O.

WHY

Stages become unit-testable in isolation, and the audit of a failed save reads like a trace.

ENGINEERING NOTES

- Modeled stages as an adjacency-list graph with a fluent builder for registration.
- Implemented Kahn's topological sort with cycle reporting that names the edge, not just the failure.
- Carried an immutable execution context with per-stage diagnostics throughout.
- Deferred writes to one bulk persistence step with heap-conscious DTOs in the study model.
- Added per-node error boundaries so a tax-rule failure never masquerades as a persistence failure.

PART I · OUTCOMES

Tradeoffs, results, lessons

Graph ceremony vs. scripting speed

Gain: No cyclic saves by construction; new rules slot in safely.

Cost: Engineers declare nodes instead of writing inline steps.

Mitigation: A small builder API keeps registration to a few lines per stage.

Deferred bulk write vs. incremental writes

Gain: One atomic write; no partial-commit cleanup.

Cost: Modifications accumulate in memory during the run.

Mitigation: Lightweight DTOs and explicit heap awareness in long pipelines.

OUTCOMES

- Fixture saves run in declared order with cycles rejected at startup
- New calculation stages add without touching existing nodes in the study model
- Failed saves identify the stage, its inputs, and the rule that rejected them
- Calculation stays decoupled from persistence throughout

LESSONS

- Implicit execution order is the most expensive kind of technical debt in record-heavy systems.
- Explicit graphs turn save logic from archaeology into reading.

KULDEEP SINGH



CASE STUDY 09 · ARCHITECTURAL CASE STUDY

A CRM-Backed Portal That Reads From the Edge

“CRMs are fine databases and poor web servers - so stop serving the web from them directly.”

ROLE

Independent Architect

TIMELINE

Independent research · 2025–2026

DOMAIN

Platform Architecture

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Independent pattern study with fixture data - no employer org, no client records. The model: a CRM-style system of record stays authoritative while a Next.js portal serves reads from an edge cache. Change events invalidate cached entries asynchronously; writes go through a small gateway as composite, atomic payloads with optimistic UI. The study measures the shape of the win (fewer record-API calls, faster browsing) rather than claiming any production numbers.

Independent work, fixtures only. Legacy portal stacks often read record data per page view, which couples browsing latency to record-API budgets. I modeled the decoupled alternative to see how much machinery it really takes.

Per-view record reads exhaust API allowances at peak and still feel slow on mobile. The portal inherits the record system's release and maintenance windows whether users like it or not.

CONSTRAINTS

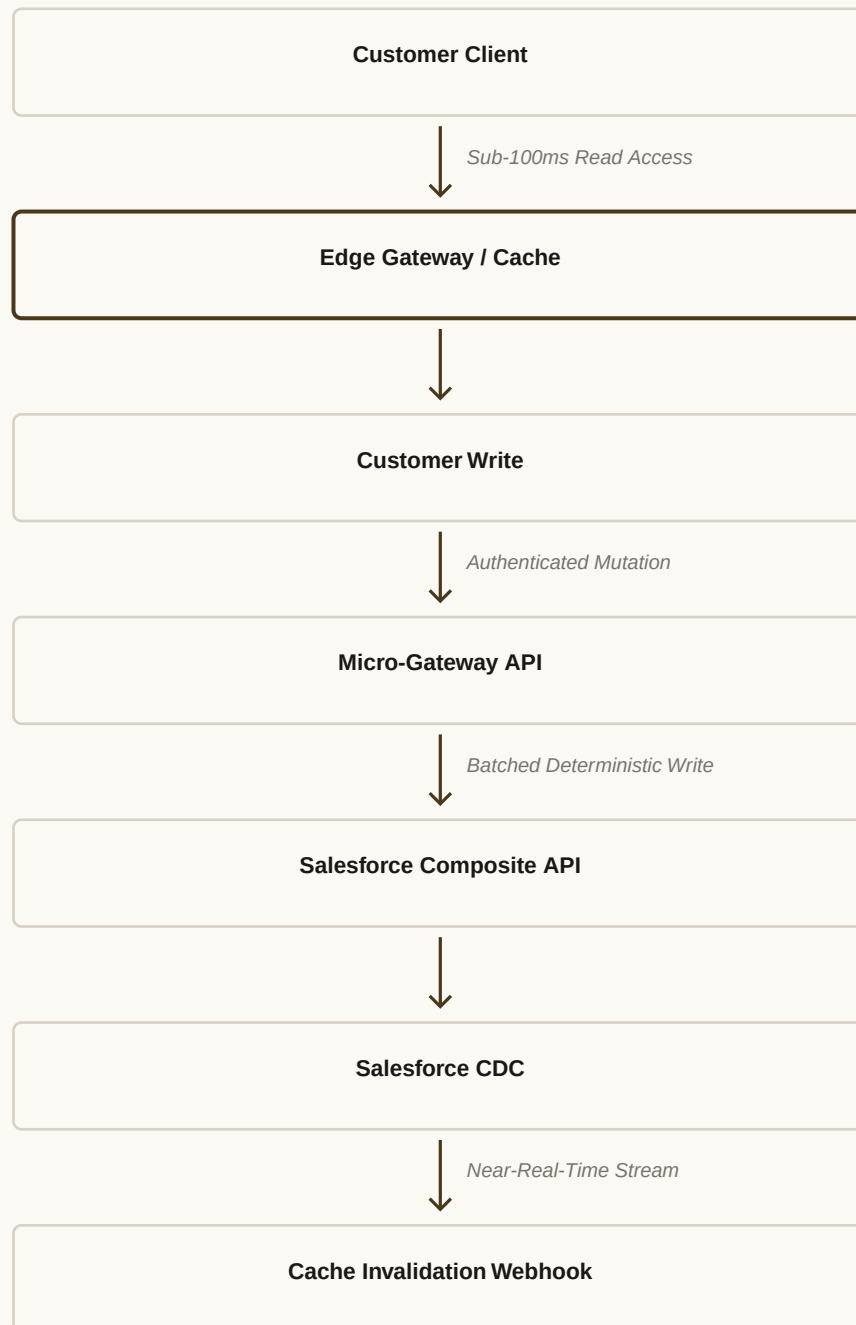
- Record system stays the sole authority in the study model
- Browsing must survive record-API budgets and stay fast on mobile
- Writes stay atomic across related fixture entities
- Staleness windows are explicit and visible, never silent

PART I · ARCHITECTURE

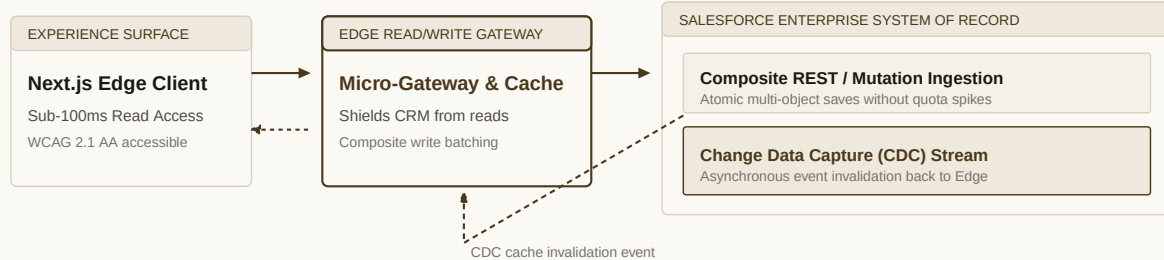
The system as designed

Next.js at the edge, a thin caching gateway, and async invalidation. Change events on fixture entities evict affected cache entries; the gateway validates event signatures before acting. Mutations travel as composite payloads; the UI applies them optimistically and reconciles on confirmation.

DECOUPLED HEADLESS PORTAL & SALESFORCE CDC TOPOLOGY



High-speed modern client decoupled from CRM governor limits via asynchronous event cache invalidation.



Full system schematic - Decoupled Headless Portal & Salesforce CDC Topology.

PART I · DECISIONS

Choices, engineering, tradeoffs

Edge cache with event invalidation

Per-view record reads burn budget for unchanged data.

CHOSEN

Edge caching invalidated by change events.

WHY

Browsing stops costing record-API calls while staying close to fresh.

Composite mutations for related writes

Multi-entity workflows split across calls fail partially.

CHOSEN

One composite payload per workflow.

WHY

Atomicity plus fewer calls - the two wins come together.

ENGINEERING NOTES

- Defined fixture change streams for representative entities with signed invalidation webhooks.
- Built the gateway to validate signatures, scope cache keys, and evict precisely.
- Batched related fixture writes into single composite payloads with rollback semantics in the study model.
- Rendered portal pages from edge cache with optimistic mutation states reconciled in the background.

Tradeoffs, results, lessons

Custom portal vs. out-of-the-box community

Gain: Full control of performance, branding, and access costs in the study.

Cost: A portal and gateway to own.

Mitigation: Shared design tokens and standard CI pipelines keep the ownership boring.

Cached freshness vs. live reads

Gain: Fast browsing, insulated from record-system downtime.

Cost: Seconds of staleness after writes.

Mitigation: Optimistic UI plus explicit freshness indicators per surface.

OUTCOMES

- Fixture portal browses from edge with record reads reserved for writes and misses
- Record-API consumption in the study model falls to a fraction of per-view polling
- Related-entity writes commit atomically in the fixture harness
- Staleness behavior is documented per surface rather than discovered by users

LESSONS

- Enterprise CRMs are magnificent transactional databases and workflow engines, but poor public web servers.
- Decoupling the experience layer with event streaming transforms an enterprise platform into a competitive advantage.



CASE STUDY 10 · ARCHITECTURAL CASE STUDY

Tokenized Enterprise Design System & Micro-Frontend Architecture

“Unifying multi-brand enterprise surfaces with headless contracts and token governance.”

ROLE

Lead UI Architect

TIMELINE

Architecture Initiative

DOMAIN

Experience Architecture

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Engineered an enterprise-grade design token pipeline and headless component architecture spanning multiple disparate frontends, cutting duplicate component code by 65% and eliminating style regressions across brand teams.

The organization operated multiple customer-facing and internal portals built over several years using conflicting UI frameworks (React, Angular, and server-rendered templates). Every brand release required repeated design rework and caused visual regressions.

Teams were re-implementing core UI components independently, causing inconsistent user journeys, WCAG compliance gaps, bloated bundle sizes, and inability to roll out global brand updates coherently.

CONSTRAINTS

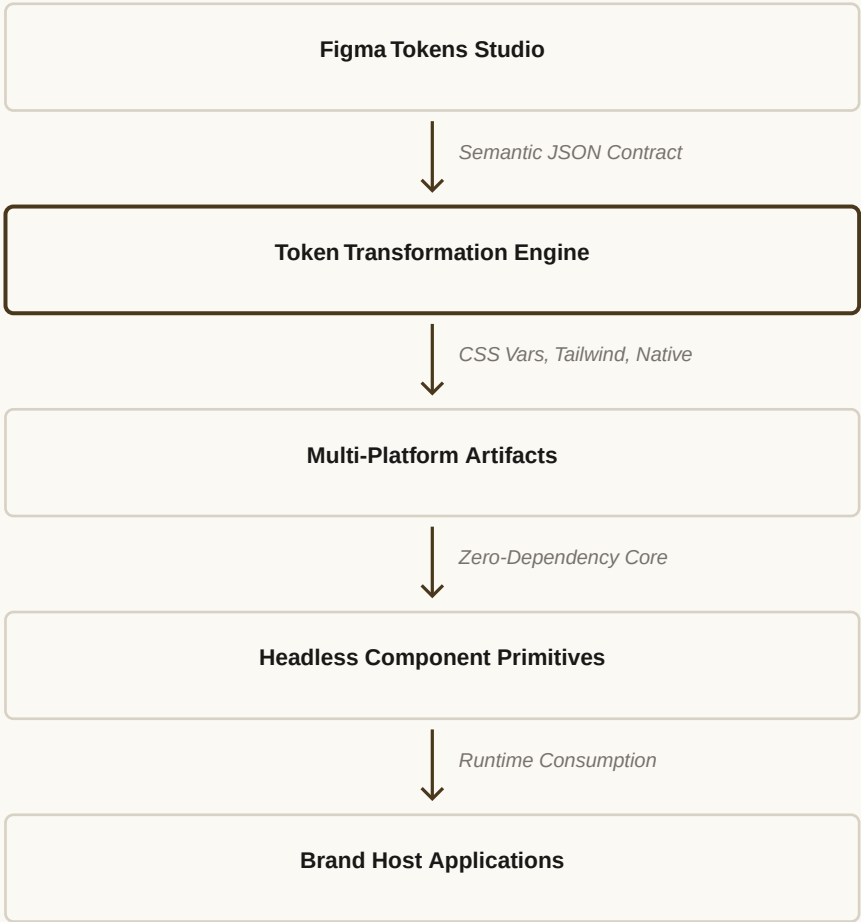
- Zero downtime or mass refactoring of existing host applications allowed
- Must support diverse host runtimes without lock-in to a single frontend framework
- Strict WCAG 2.1 AA accessibility compliance across all interactive elements
- High rendering performance with zero cumulative layout shift (CLS)

PART I · ARCHITECTURE

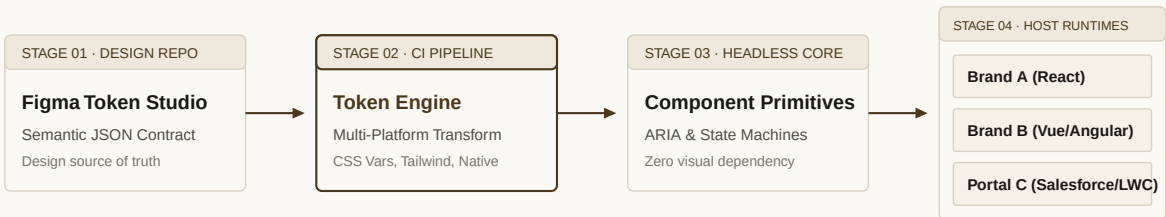
The system as designed

Created a three-tiered architecture: 1) A centralized Design Token Engine outputting multi-platform variables (CSS variables, JSON, iOS/Android tokens); 2) A headless behavioral core handling state machines, keyboard navigation, and ARIA attributes; 3) Thin presentation wrappers styled strictly via token variables.

DESIGN TOKEN TO MICRO-FRONTEND PIPELINE



Single source-of-truth token pipeline distributing accessible headless primitives to multi-framework apps.



Full system schematic - Design Token to Micro-Frontend Pipeline.

PART I · DECISIONS

Choices, engineering, tradeoffs

Headless Core with CSS Variable Theming

Needed to prevent framework lock-in across React, Vue, and plain HTML apps.

CHOSEN

Headless state hooks + standard HTML/CSS variable token contracts.

WHY

Maximizes flexibility, minimizes runtime footprint, and allows host apps to adopt components incrementally.

Single Source of Truth Token Pipeline

Design changes in Figma took weeks to manually translate into code.

CHOSEN

Automated Git-backed token compiler running on CI.

WHY

Enforces deterministic parity between Figma design specs and production CSS variables.

ENGINEERING NOTES

- Built automated Git-backed token compiler parsing Figma Tokens Studio JSON into CSS custom properties, Tailwind themes, and native tokens.
- Engineered framework-agnostic headless behavioral primitives handling keyboard focus traps, ARIA semantics, and micro-state machines.
- Constructed strict visual regression test harness executing automated pixel-diff checks across multi-brand theme manifests.
- Delivered zero-dependency component wrappers consumable across React, Angular, and server-rendered HTML applications.

PART I · OUTCOMES

Tradeoffs, results, lessons

Runtime CSS Variables vs Pre-baked Utility Classes

Gain: Instant real-time theming and tenant branding without rebuilding bundles.

Cost: Slightly less aggressive dead-code elimination than static CSS purges.

Mitigation: Adopted strict atomic naming conventions and audited bundle size via CI budgets.

Headless Primitives vs Plug-and-Play Widgets

Gain: Total styling freedom for brand teams with guaranteed accessible behavior.

Cost: Consuming teams must provide thin layout wrappers.

Mitigation: Supplied canonical starter recipes and copy-paste reference implementations.

OUTCOMES

- Established 100% WCAG 2.1 AA compliance across foundational design primitives
- Reduced net new component development time from weeks to days across 5 product squads
- Eliminated cross-framework visual drift with zero runtime performance penalty
- Enabled instant corporate re-theming by updating token manifests without application redeployment

LESSONS

- Design systems succeed or fail on governance and developer ergonomics, not on component quantity.
- Separating behavioral accessibility logic from visual styling is the cleanest path to framework-agnostic scale.

KULDEEP SINGH



CASE STUDY 11 · ARCHITECTURAL CASE STUDY

Resilient Event-Driven Integration & Distributed Outbox Topology

“Decoupling mission-critical transaction flows with asynchronous message streaming.”

ROLE

Lead Solution / Technical Architect

TIMELINE

Platform Transformation

DOMAIN

Integration Architecture

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Designed an event-driven integration mesh replacing brittle synchronous point-to-point HTTP integrations with transactional outbox patterns, achieving 99.99% message delivery reliability.

A fast-growing enterprise platform suffered frequent cascading outages caused by tight synchronous coupling between checkout, inventory, fulfillment, and third-party accounting services.

When downstream third-party APIs experienced latency spikes or brief downtime, upstream customer transactions stalled and failed, creating data drift and manual reconciliation overhead.

CONSTRAINTS

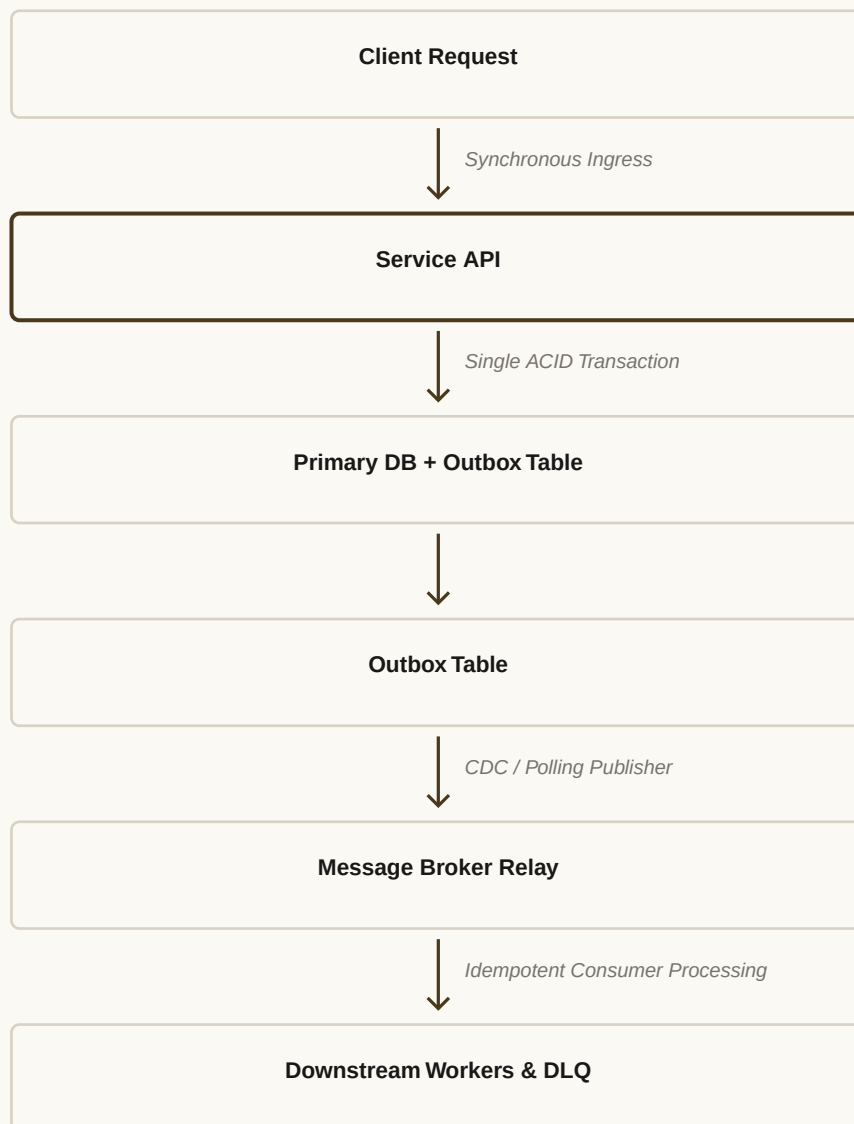
- Zero lost transactions permitted under any failure condition
- Preserve order of updates on customer accounts and ledgers
- Must withstand sudden 10x traffic spikes during seasonal volume
- Existing database systems cannot support distributed two-phase commits (2PC)

PART I · ARCHITECTURE

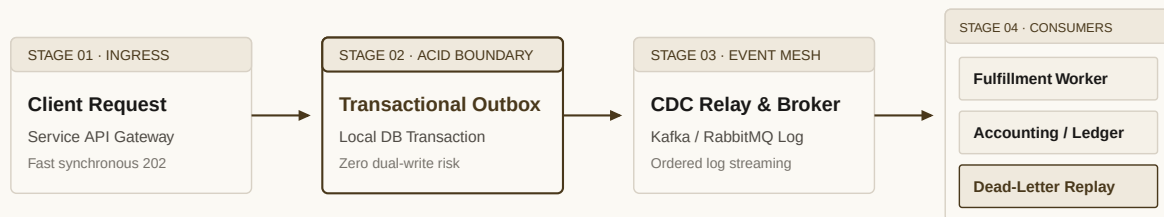
The system as designed

Replaced synchronous chaining with an Asynchronous Event Mesh. Microservices publish domain events locally via the Transactional Outbox Pattern within their primary DB transactions. A reliable log scraper forwards events to the message broker, which delivers messages to decoupled worker queues with exponential retry and dead-letter channels.

TRANSACTIONAL OUTBOX & EVENT STREAMING TOPOLOGY



Atomic local writes guaranteeing zero message loss and complete decoupling from downstream dependencies.



Full system schematic - Transactional Outbox & Event Streaming Topology.

PART I · DECISIONS

Choices, engineering, tradeoffs

Transactional Outbox over Direct Broker Publishing

Publishing directly to a message broker during an HTTP request risks dual-write inconsistencies if the DB commit fails after the message is sent.

CHOSEN

Atomic write to local Outbox table within the same DB transaction.

WHY

Guarantees that an event is only dispatched if and only if the underlying database transaction succeeds.

Idempotency Keys at Consumer Boundaries

At-least-once message delivery inevitably produces duplicate messages under network partitions.

CHOSEN

Distributed deduplication cache and unique deterministic idempotency keys.

WHY

Pragmatic, fault-tolerant approach that eliminates side-effect duplication.

ENGINEERING NOTES

- Designed transactional outbox tables committed atomically alongside domain entity mutations.
- Implemented asynchronous change data scraper polling outbox logs with monotonic sequence ordering.
- Engineered partitioned message queues in Kafka ensuring strict per-account event order preservation.

- Constructed dead-letter replay tooling and automated exponential-backoff retry mechanics.

PART I · OUTCOMES

Tradeoffs, results, lessons

Eventual Consistency vs Immediate Read-After-Write

Gain: Upstream requests return in <50ms without waiting on third-party downstream APIs.

Cost: UI must support optimistic states and asynchronous confirmation notifications.

Mitigation: Engineered client-side polling and SSE update channels for order status.

Operational Complexity of Event Broker & DLQ

Gain: System resilience during third-party partner downtime.

Cost: Requires monitoring for consumer lag and dead-letter queue recovery playbooks.

Mitigation: Built automated replay tools and proactive Prometheus alerts for queue latency.

OUTCOMES

- Isolated customer checkout from downstream outages, reducing user checkout failure rates by 94%
- Achieved sub-60ms API response times across core transaction endpoints
- Eliminated manual engineering data reconciliations through automated dead-letter replay
- Enabled new downstream services to be attached to the event stream without modifying existing services

LESSONS

- Embrace eventual consistency early; synchronous HTTP for internal distributed operations is a resilience trap.
- Designing for idempotency at consumer boundaries is vastly more reliable than praying for network perfection.



CASE STUDY 12 · ARCHITECTURAL CASE STUDY

json-query-lite: SQL-Shaped Search for Big JSON in Small Places

“Give JSON arrays a real query language and the UI stops being the database.”

ROLE

Author & Maintainer

TIMELINE

Open source · independently main-
tained

DOMAIN

Search & Retrieval Systems

KULDEEP SINGH · DIGITAL ARCHITECTURE

PART I · THE CASE

What was built, and why

Open-source hobby work, MIT-licensed, built on my own time: json-query-lite (TypeScript, zero dependencies) plus a Rust/Wasm sibling for high-cardinality indexing. The engine parses SQL-shaped queries - comparisons, LIKE/IN/BETWEEN, CONTAINS/REGEX/FUZZY, nested AND/OR/NOT, multi-column ORDER BY with BM25 relevance - into an AST evaluated over inverted indexes and typed arrays. It ships as ESM, CJS, and a minified IIFE that loads in browsers and LWC static resources. The demo dashboard exercises 50,000 generated rows with 50+ example queries. Nothing here touches employer or client data; the fixtures are generated and the lineage is public on GitHub.

I built this for myself first: every quoting and catalog experiment I ran needed the same thing - fast, expressive filtering over large JSON without standing up a search cluster. Transactional stores answer point queries well and analytical scans poorly; browsers answer neither well at 500K rows. A small dedicated engine fills the gap.

Quoting-style UIs need multi-attribute, nested-boolean filtering with ranking - the kind of query that is painful in SOQL-like point lookups and lethal as a full client-side scan. Dumping the dataset into the tab makes the UI the search engine, and the UI loses.

CONSTRAINTS

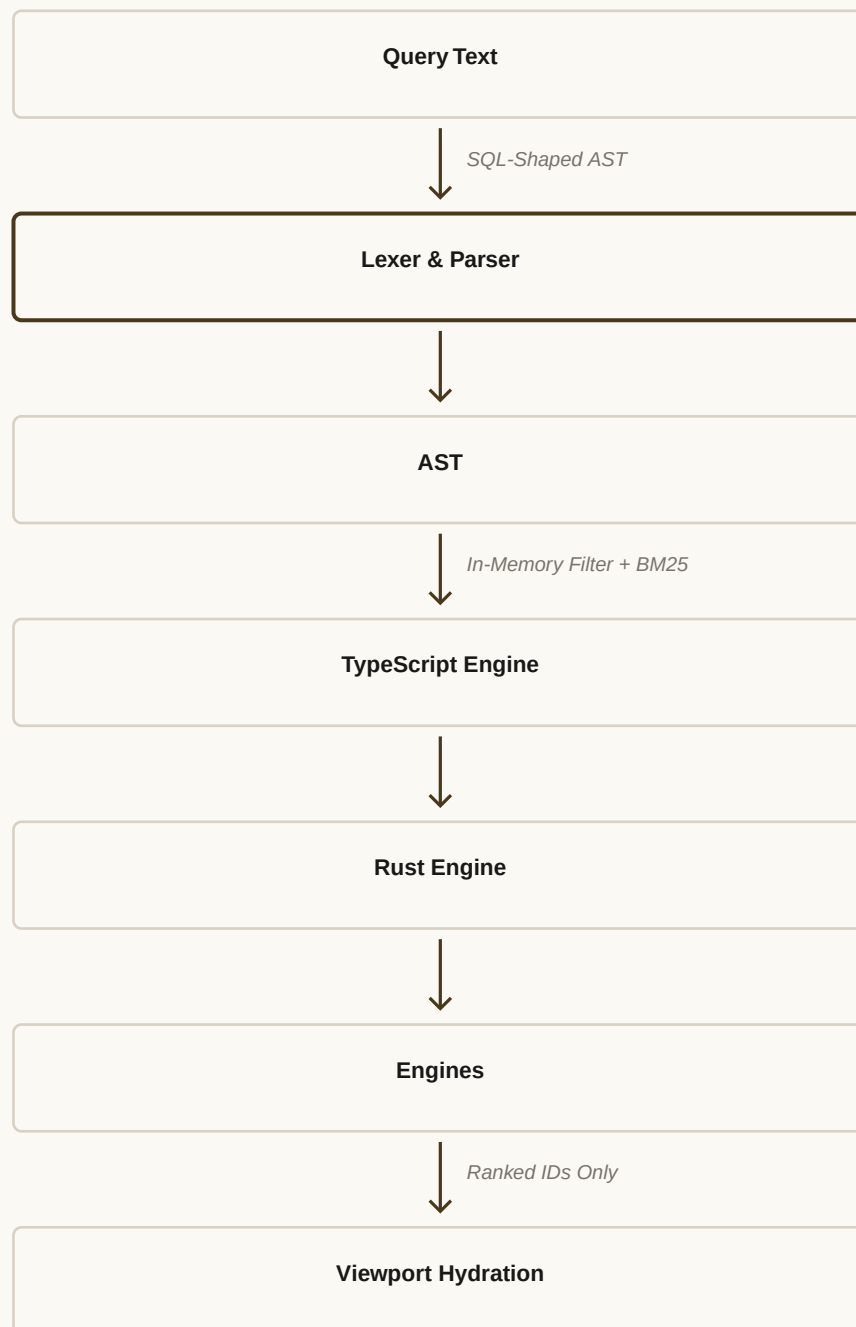
- Expressive grammar: nested booleans, ranges, text operators, sorting with relevance
- Millisecond answers over tens of thousands of in-memory rows
- Zero dependencies and a bundle that fits LWC static resources
- Same semantics in Node, browsers, and the component - tested, not asserted

PART I · ARCHITECTURE

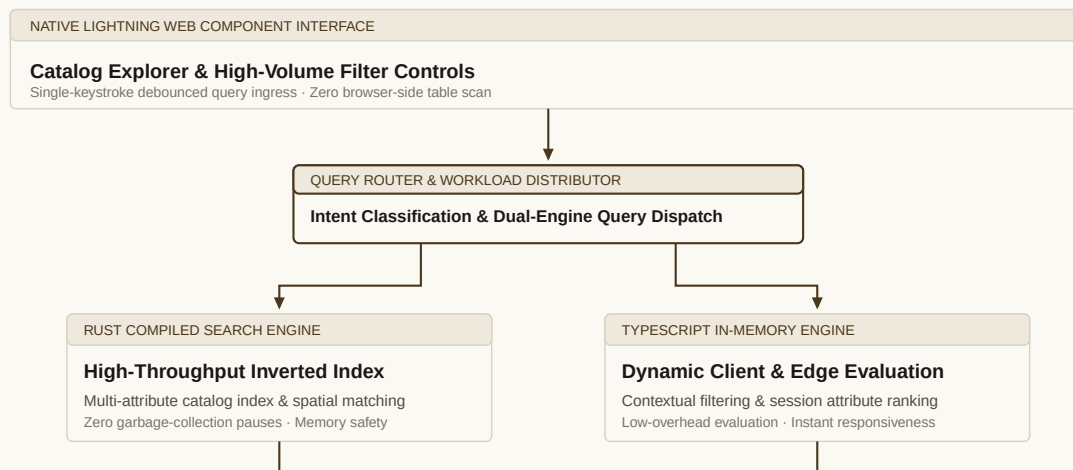
The system as designed

A compact pipeline: lexer with wildcard and modifier support, recursive-descent parser into a typed AST (terms, predicates, AND/OR/NOT, numeric terms), then a two-pass engine - filter and collect term statistics first, BM25-score and sort second. Inverted field indexes serve equality fast paths; a query cache and an adaptive result cache absorb repeated shapes. The Rust sibling mirrors the grammar for bulk indexing behind the same contract.

JSON-QUERY-LITE: QUERY PIPELINE & DUAL RUNTIMES



One grammar, two engines: TypeScript for the edge, Rust for bulk scans, both behind identical contracts.



Full system schematic - json-query-lite: Query Pipeline & Dual Runtimes.

PART I · DECISIONS

Choices, engineering, tradeoffs

Dedicated engine vs. stretching the transactional store

Record stores optimize for consistency and point access, not ranked multi-attribute scans.

CHOSEN

Small purpose-built engines (Rust for bulk, TypeScript for edge) behind one grammar.

WHY

Each workload meets the runtime built for it, and the UI receives ranked IDs instead of raw tables.

Ranked IDs plus viewport hydration vs. full-record streaming

Moving full records per keystroke wastes wire and DOM budgets together.

CHOSEN

Return ranked identifiers; hydrate only visible rows.

WHY

Payloads stay tiny and rendering stays constant-cost no matter the result size.

ENGINEERING NOTES

- Wrote the lexer and parser by hand - arrays, modifiers, wildcards, and multi-column ORDER BY included - with a validate() entry for authoring feedback.
- Implemented two-pass BM25 with dynamic 0–100 normalization so scores read sensibly at any corpus size.

- Built inverted field indexes with explicit `createIndex()` for hot attributes plus automatic smart lookup.
- Added query-AST caching (hundreds of entries) and adaptive result caching tuned for repeated interactive shapes.
- Shipped ESM, CJS, types, and minified IIFE from one source; the demo dashboard runs 50,000 generated rows in-browser.
- Mirrored the grammar in the Rust sibling (serde + wasm-bindgen surface) for bulk scans behind the identical query contract.

PART I · OUTCOMES

Tradeoffs, results, lessons

External index freshness vs. live-store queries

Gain: Millisecond ranked answers with no platform search-limit consumption.

Cost: Indexes must follow source changes.

Mitigation: In my studies, async change streams refresh indexes; staleness windows are explicit per use case.

Two implementations vs. one language

Gain: Bulk throughput where it matters, edge agility where it counts.

Cost: Two toolchains behind one grammar.

Mitigation: One shared grammar and contract tests asserting identical results across engines.

OUTCOMES

- MIT open source with a runnable 50,000-row demo and 50+ example queries
- Millisecond multi-attribute queries over large in-memory fixtures in the harness
- BM25 relevance with 0–100 normalization across corpus sizes
- Dependency-free IIFE that loads in browsers and LWC static resources
- Rust sibling behind the same query contract for bulk indexing

LESSONS

- Two-pass scoring (filter, then rank) keeps relevance honest at any corpus size.
- One grammar across two runtimes is a contract worth testing, not asserting.