



CASE STUDY · 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

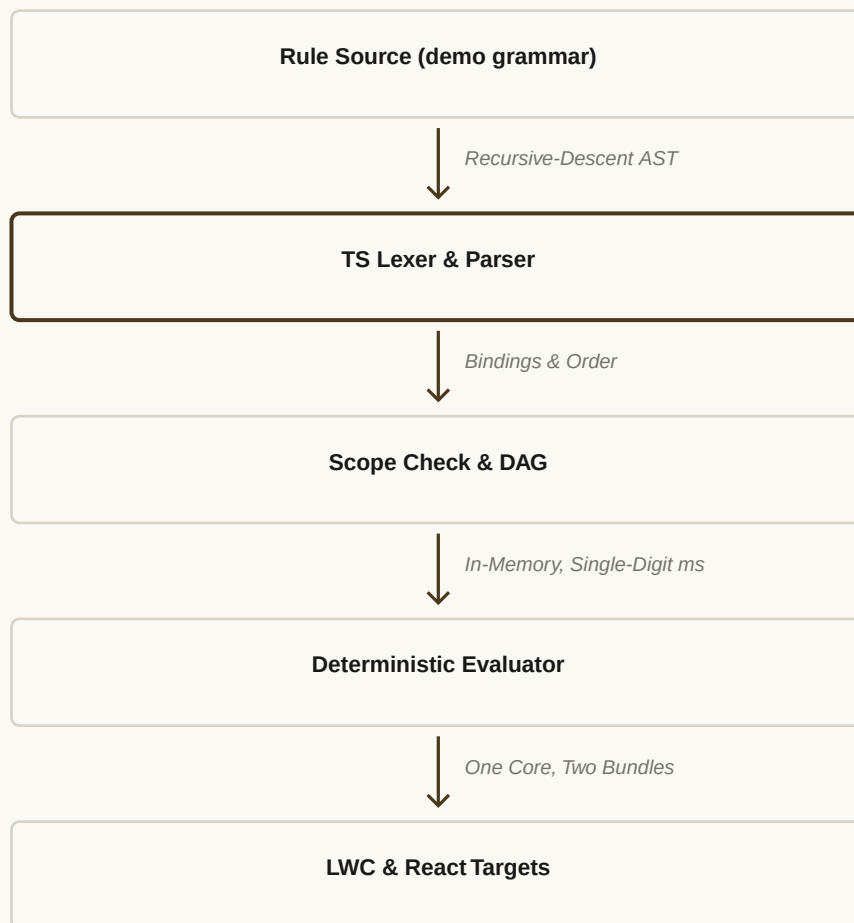
PART I · ARCHITECTURE

The system as designed

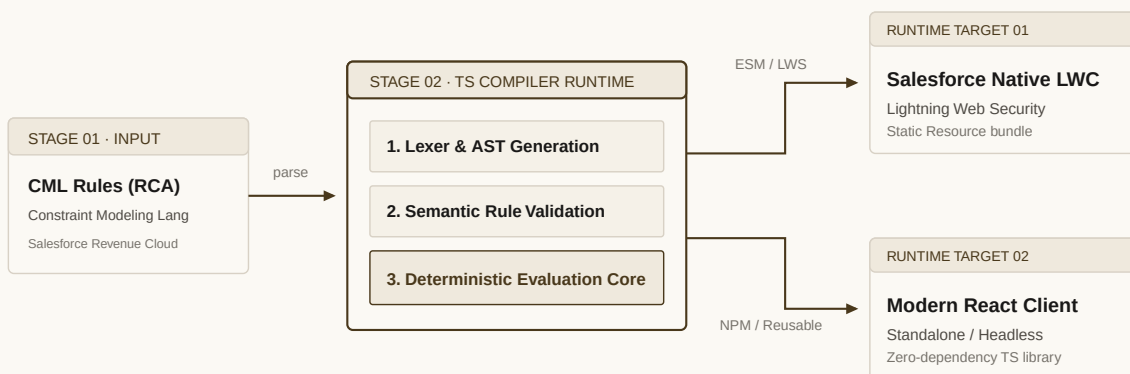
Four stages, all in plain TypeScript, no dependencies:

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.

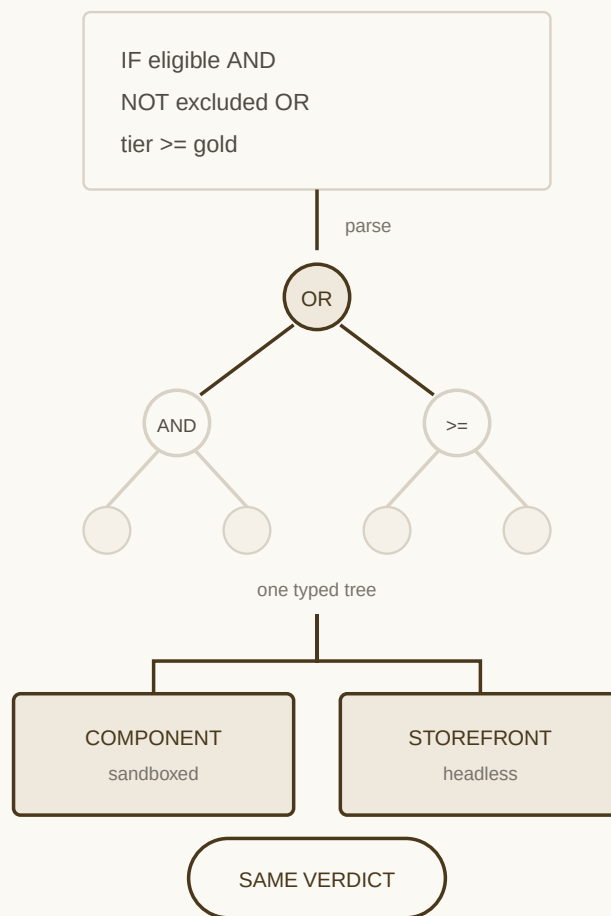


Exhibit plate - the system at a glance.

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

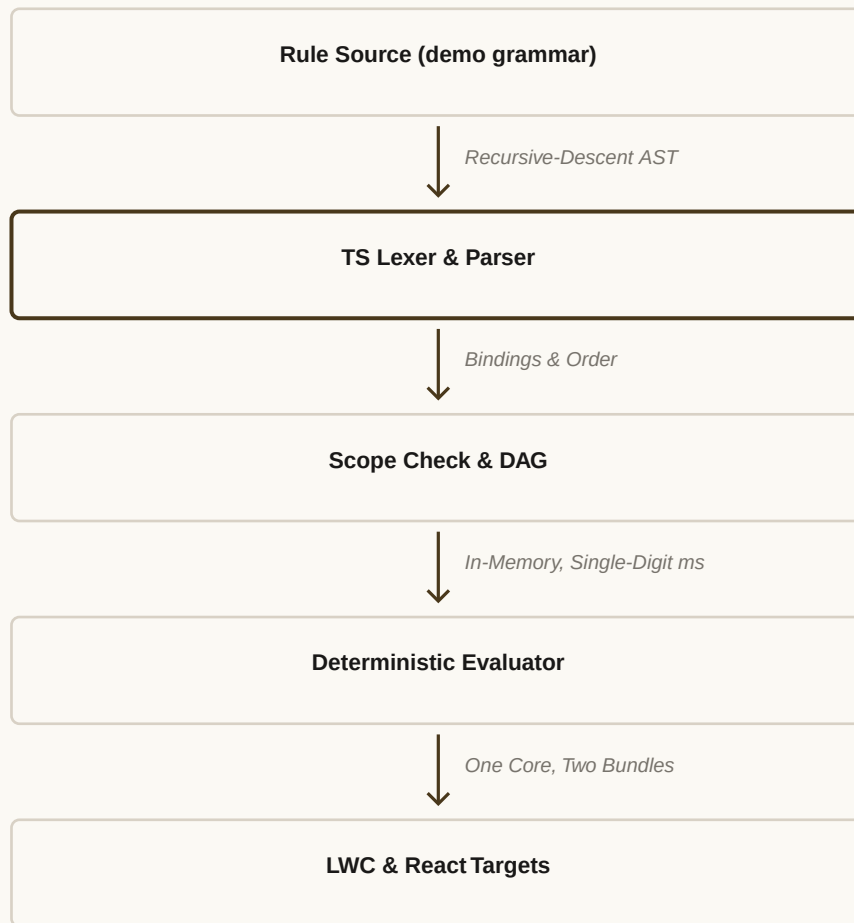
03 · THE ARCHITECTURE

Inspect the system

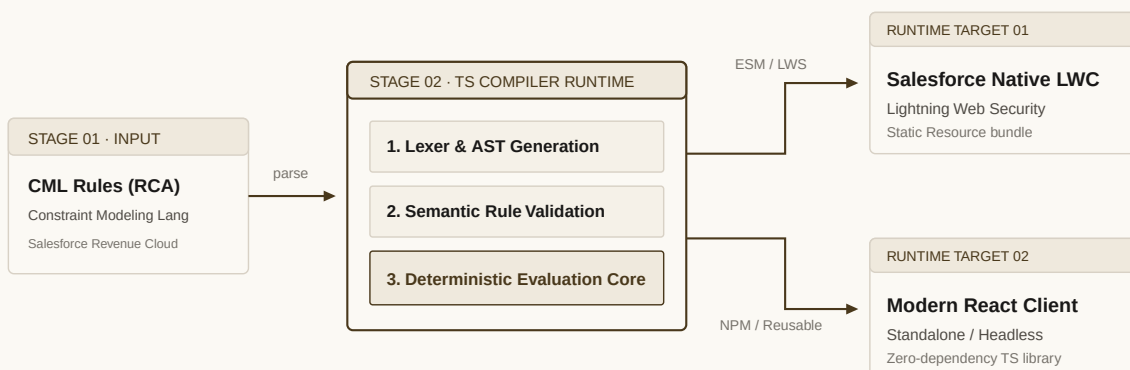
From rule text to evaluated result through a real language pipeline.

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

More in the Deep Dive Series

- [01 · Generic Entity Scoring Engine, Native to the CRM](#)
- [02 · BrewCMS: Open-Source Content Control Plane & Agent Operating System](#)
- [03 · Local Evaluation of Salesforce RCA - CML](#)
- [04 · TM Forum Open API Intelligence Suite & Spectra AI Copilot](#)
- [05 · WebAssembly in Salesforce LWC: 500K Record Engine & Net-Zero DB](#)
- [06 · Multi-Site Quoting Without Leaving the CRM](#)

Continue the exploration

- [Interactive exhibit: victorkuldeep.com/case-studies/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

