CASE STUDY · 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

PART I · THE CASE

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).

KULDEEP SINGH

· 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.

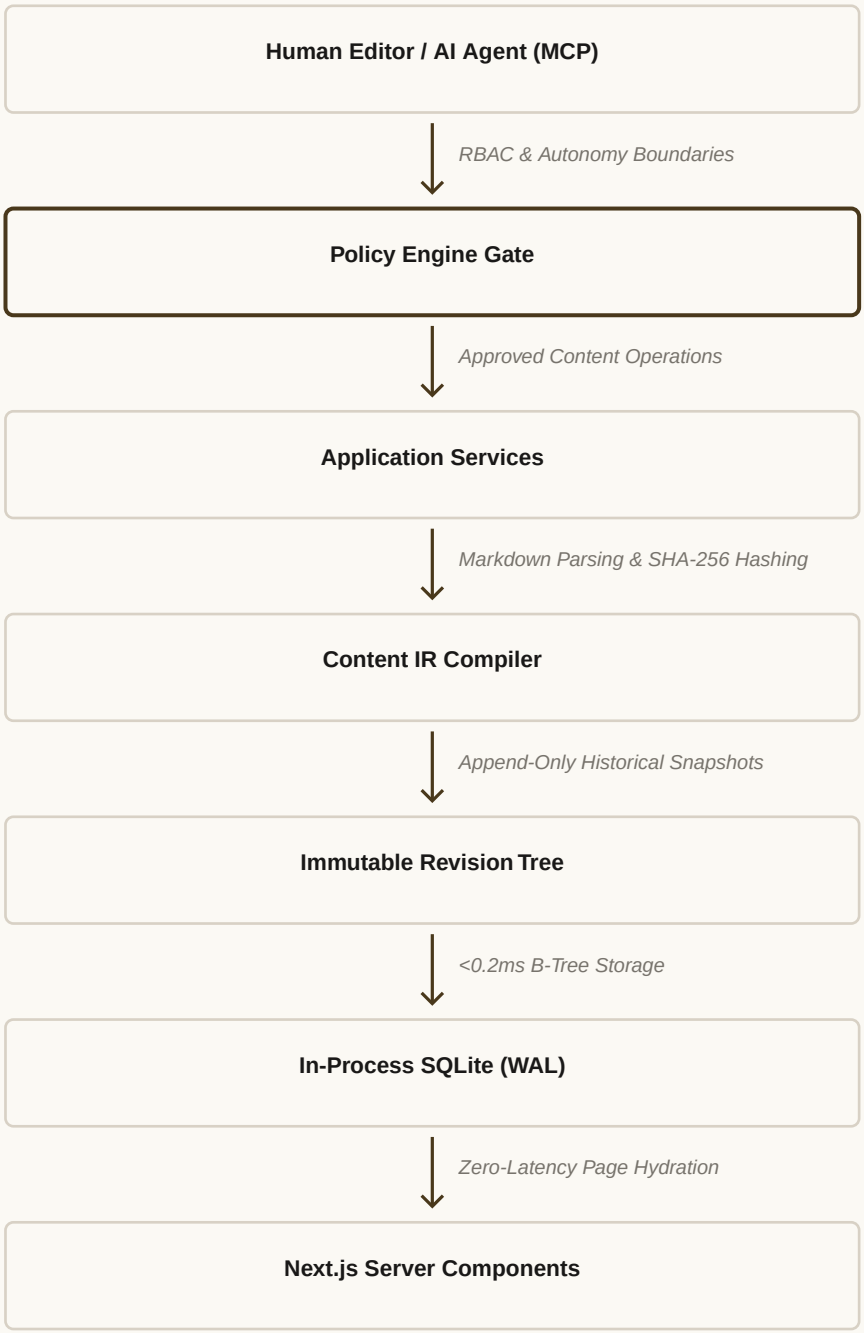
PART I · ARCHITECTURE

The system as designed

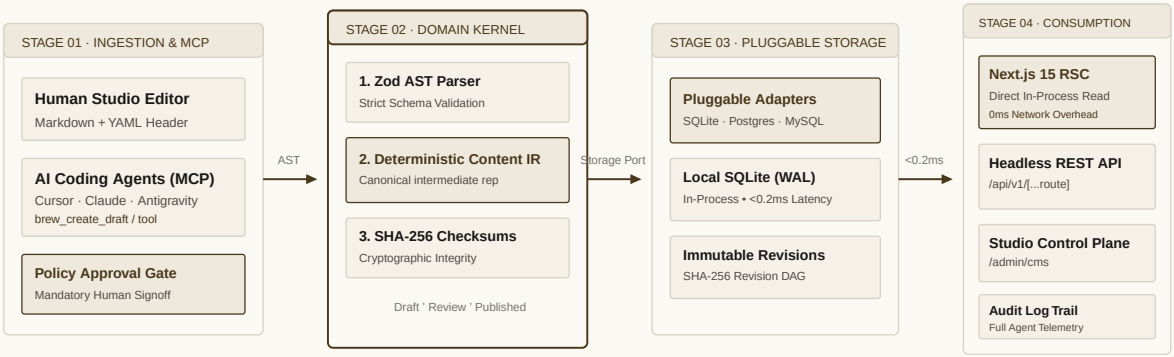
BrewCMS implements a strict Hexagonal / Ports & Adapters architecture. At the center is @brew-cms/core, containing pure TypeScript domain entities, business logic, and application services with zero framework dependencies. The content layer (@brew-cms/content) tokenizes Markdown and Frontmatter into 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.

KULDEEP SINGH

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.

KULDEEP SINGH

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.

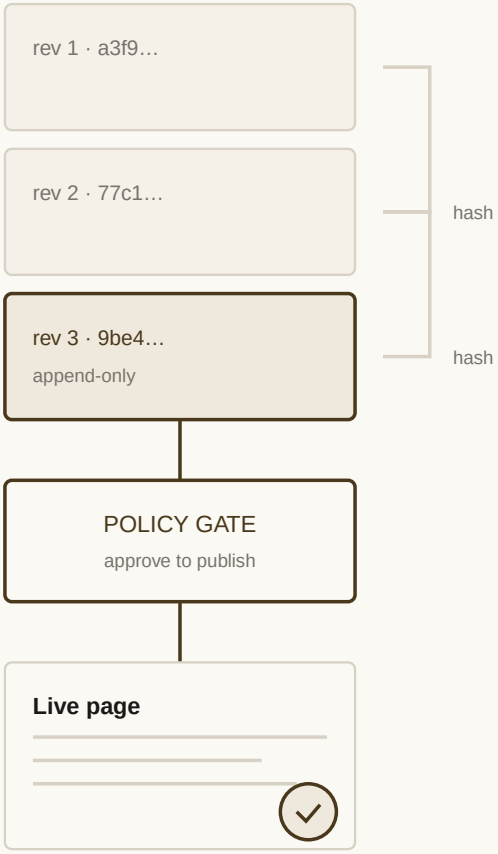


Exhibit plate - the system at a glance.

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

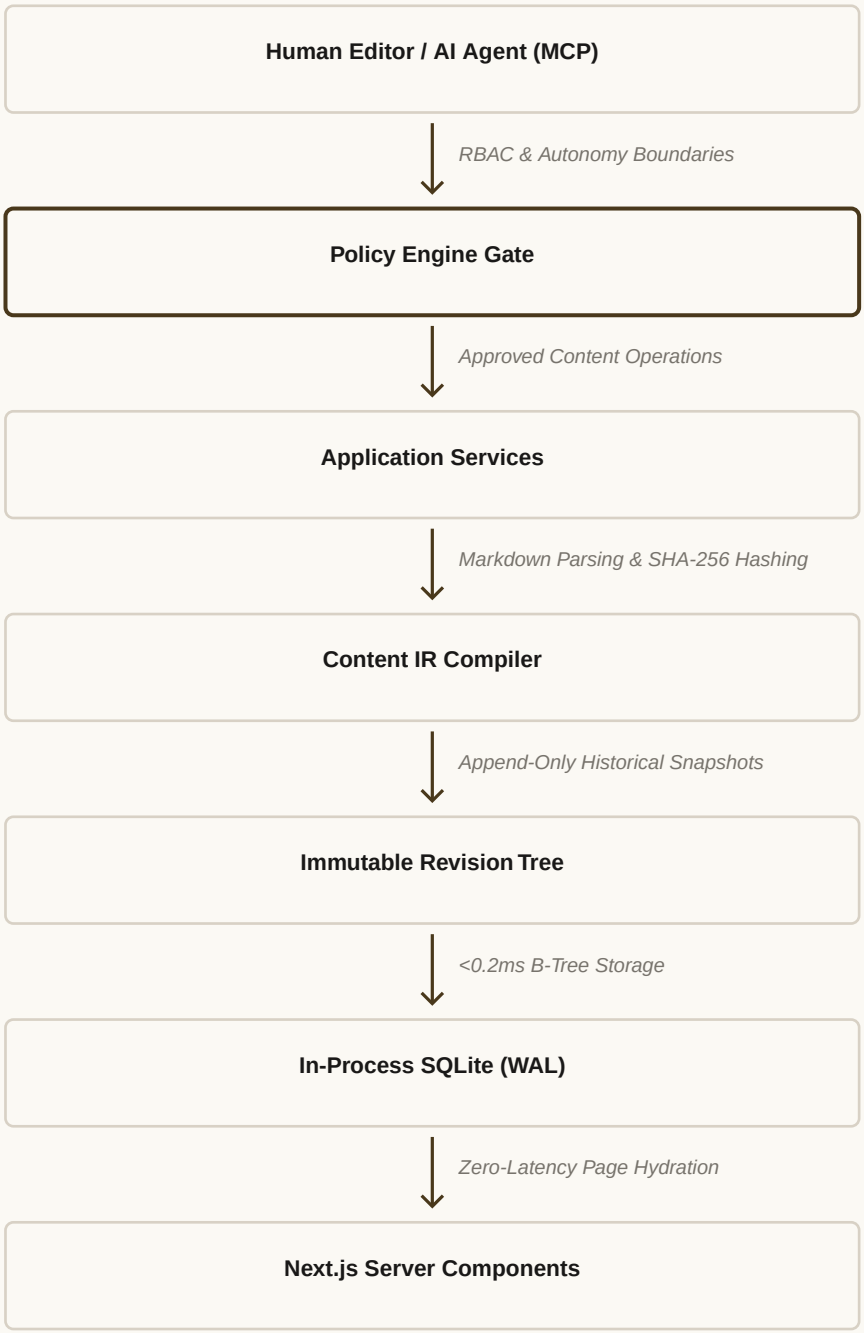
Inspect the system

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

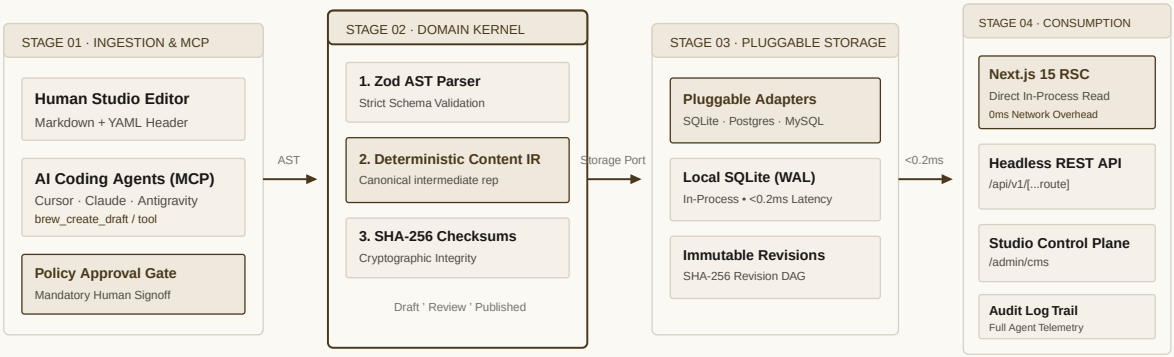
Six boundaries. Each one removes a class of surprise.

KULDEEP SINGH

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.

KULDEEP SINGH

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.

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 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.

KULDEEP SINGH

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/brew-cms/deep-dive

Long-form essays: victorkuldeep.com/thinking

Ask AXIOM: victorkuldeep.com/ask

Connect: victorkuldeep.com/connect

Kuldeep Singh

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

KULDEEP SINGH