Correlation Studio — Technical Whitepaper

Rapid discovery mining and causation analysis. Data science without the code.

Matthew Meadows (Rango) · October 2026 · correlationstudio.com

The Premise

Every dataset is hiding something. The relationships are in there — commodity prices tracking input costs, market indices moving with macro indicators, temperatures pairing with temperatures a watershed away — but finding them traditionally requires a notebook, a language, a statistics library, and the patience to test hypotheses one at a time.

Correlation Studio inverts that workflow. You don't bring a hypothesis; you bring data. The platform mines **every numeric column pair** across your datasets for statistically significant relationships, ranks what it finds, tests it for predictive causality, writes up the analysis, and gives every result a permanent, shareable, citable page. No Python. No notebooks. No code.

This document is the technical story of how that works: the architecture, the algorithms, the statistics, and the engineering arc that got it here — including the version that broke, and the rebuild that didn't. It is, end to end, the work of one engineer over roughly nine months and some 850 commits. That constraint isn't a footnote; it's the design principle. Every architectural decision below is biased toward *simplicity at the expense of optionality* — one database, one app server, one object store, one embedded query engine — because a system simple enough for one person to fully understand is a system one person can operate, debug, and evolve at production speed.

What It Is

Correlation Studio is a vertically-integrated correlation-analysis platform. The workflow, end to end:

1. **Bring data in.** Upload CSV, TSV, Excel, or Google Sheets; paste raw text; supply URLs; or describe what you're looking for and let three AI providers — Claude, Gemini, and Grok — search the open web for sources in parallel. Every discovered URL is verified, downloaded, type-detected, and previewed before ingestion — and a dataset sourced from a URL can keep itself current, re-checking its source on a schedule.
2. **Pair datasets into Experiments.** Choose a comparison mode (one-against-one through everything-against-everything), a join strategy (by date, by shared key, by shared geography, or by row order), and a significance threshold.
3. **Mine.** The engine computes correlations across every numeric column pair. Each pair whose strength clears the threshold becomes a **Discovery** — a first-class entity carrying Pearson and Spearman coefficients, a p-value, a 95% confidence interval, an interactive chart, and drill-down access to the exact source rows behind any datapoint.
4. **Interrogate.** Ten visualization modes, five regression families with prediction intervals, lag analysis, rolling correlation, and on-demand Granger causality testing in both directions.
5. **Explain.** AI analysis reads the statistics like a senior analyst — strength, direction, confounders, caveats, actionable insights — and pins a written narrative to the entity.
6. **Publish.** Discoveries, experiments, datasets, and block-composed Portfolios can be published to a public feed, rated, discussed, and shared. Every public entity gets a stable URL, structured metadata, and search-engine-grade rendering.
7. **Ask.** Corrie, the in-app assistant, answers questions against the entire public corpus with real coefficients and citations that link back to the source discoveries.

The corpus

As of early October 2026, the public corpus stands at roughly:

Public entity	Count

Discoveries	**55,000**
Experiments	**9,300**
Datasets	**7,700**

Every public discovery carries real statistical context *and* a written analysis — the corpus was seeded deliberately so that search, the chatbot, and human browsers all retrieve substance, not just titles and r-values. The entire public corpus is published as **open data under CC BY 4.0**, with creator attribution, in machine-readable form (more on that below). It grows through a systematic, politeness-first crawler that harvests open-data portals — and since October it also keeps itself current: datasets re-check their sources on a schedule, and the experiments built on them recompute in place.

The Architecture

The production system is deliberately small:

- A **single PostgreSQL 17 database** for all metadata: users, billing, datasets, experiments, discoveries, jobs, audit, and the RAG vector index (via `pgvector`).
- **Cloudflare R2 object storage** for all bulk data, stored as columnar **Apache Parquet** — one file per dataset, one small file per discovery's chart payload. Per-byte pricing, zero egress fees.
- **DuckDB embedded in the application server**, querying Parquet directly through a local NVMe hot-tier cache with LRU eviction. Analytical SQL runs in-process; there is no separate analytics cluster.
- A **.NET 10 ASP.NET Core API** in a clean three-layer solution, hosting all parsing, correlation math, AI orchestration, thumbnail rendering, billing, and two dozen background services.
- A **React 18 + TypeScript SPA** with canvas-rendered charts, route-level code splitting, and optimistic UI throughout.
- **Stripe** end-to-end for payments; **three AI providers in parallel** (Anthropic, Google, xAI) for search and analysis, plus OpenAI and Gemini for embeddings.

Two mid-range VPS instances run the whole thing — one for Postgres,

one for the app — with R2 as the third leg. That's the entire footprint.

Why this shape? Because the first shape broke.

The Crucible: Breaking at Ten Users

Commit zero was December 29, 2025. The original architecture (v1) was conventional for a data product: parsed rows landed in Postgres tables — a row store, plus two index tables holding pre-parsed join keys and numeric values per column, sharded across per-user content databases. It worked in development. It demoed beautifully.

It **fell over at roughly ten concurrent active users**.

The failure mode was write amplification. A wide CSV — a thousand columns by a few million rows — exploded into hundreds of millions of B-tree index inserts. On SATA-backed storage, concurrent ingestion turned that amplification (measured at 50–150× on wide datasets) into IOPS queue starvation: the write-ahead log couldn't drain, ingestion stalled, and everything sharing the disk stalled with it. The system hit the wall precisely when the first real traffic arrived.

The response was not a tuning pass. Tuning buys margin; it doesn't change the physics. The response was a clean-room replacement of the entire bulk-data layer — and the decision to make the new substrate match the actual workload:

- **The workload is analytical.** Column statistics, correlation joins, drilldown lookups — these read a few columns across many rows. A row store reads every byte of every row to serve them; a columnar format reads only the columns asked for.
- **The data is write-once.** A dataset is ingested, then queried many times. That's the exact profile object storage plus immutable columnar files is built for.
- **Compression is free leverage.** Dictionary and run-length encoding compress low-cardinality columns 10–100×. Less storage, less I/O, faster scans.

The Lakehouse: Parquet + DuckDB

The v2 architecture — internally, the "Lakehouse" migration — landed on May 17, 2026, about twenty weeks after commit zero. Ingestion now writes **one Parquet file per dataset** to R2. Every analytical query — column statistics, experiment correlation joins, datapoint drilldowns, thumbnail renders — is DuckDB SQL executed in-process against those files, through a local NVMe cache so hot datasets read at disk speed and cold ones fetch from R2 on demand.

The numbers that matter from the cutover:

- **Roughly 5,000 lines of C# were deleted.** The sharding apparatus, the chunked index builders, the bulk-delete machinery for hundred-million-row index tables — all of it became unnecessary, because the tables it managed no longer exist.
- **The frontend never noticed.** Every dataset, experiment, and discovery API contract survived intact. The UI that ran against v1 on Friday ran against v2 the next week.
- **The wall moved.** The architecture that collapsed at ten concurrent users now absorbs about a million requests a month — with the app server's memory headroom untouched. (Operational numbers below.)

DuckDB deserves specific credit. An embedded analytical engine that reads Parquet natively, with predicate pushdown into row groups (a query that filters on a date range skips 50,000-row chunks that can't match), collapses what would otherwise be a data-warehouse deployment into a library call. A fixed connection pool serves every read path; ingestion back-pressures politely when the pool runs hot.

The row-group structure also gives ingestion a natural batch size: rows buffer in 50,000-row groups, which is granular enough for pushdown without bloating file metadata. A typical user dataset — a hundred thousand to a few million rows — lands as a single file of 2 to 100 row groups.

Getting Data In

Ingestion is where a no-code tool earns its keep, because real-world data is hostile.

Three ways in, plus a crawler

Upload or paste covers the file-in-hand case: CSV, TSV, Excel, Google Sheets, and HTML tables, streamed to the server (files up to 100 GB) rather than buffered.

Web links accepts URLs directly and downloads server-side, with a politeness layer: per-domain concurrency caps, minimum request spacing, and honor for `Retry-After` headers.

AI Remote Search is the differentiator. Describe a topic — "daily historic S&P 500 data" — and Claude, Gemini, and Grok search in parallel for open-data sources. Results are merged and deduplicated, with URLs surfaced by multiple providers ranked higher. Every candidate URL is then **verified** before it's offered: a HEAD probe (with GET fallback) confirms the link is alive and actually serves data rather than an HTML landing page. LLMs fabricate URLs; the verification layer is what makes the feature trustworthy. Verified sources accumulate against per-user Topics, so the next search on the same subject can reuse the cache at zero AI cost.

Downloads themselves are engineered for the long tail: resumable by byte range across restarts, idle-timeout-based liveness detection (a 25-minute government-server download is normal; a wire silent for 60 seconds is not), and truncation detection for chunked-transfer endpoints that never declare a length — including a post-download ranged probe that asks the server how big the file *should* have been.

The newest addition (v2.5.0, July 2026) is a **crawler** for systematic harvest: point it at an open-data portal with a topic and seed links, and it walks the site — strictly inside `robots.txt` rules, with per-domain pacing under a dedicated user agent (`corriebot`) — surfacing every downloadable dataset it finds. A resolver stack handles the reality that download links hide behind JavaScript: schema.org `Dataset` JSON-LD (how data.gov exposes files), site-specific adapters for Socrata and FRED-style portals, and an opt-in headless renderer for fully client-rendered pages. Results hand off to the same wizard pipeline as everything else.

Parsing hostile files

Real-world CSVs open with disclaimers, contact information, and blank

lines. Headers span two or three rows. Agencies insert section banners mid-table. The preamble/header detector went through **six iterations**, each triggered by a single real-world file that broke the previous version. The current algorithm runs a two-pass analysis over the first ~50 lines — field-count consistency streaks, fill ratios, alphabetic-content detection, richest-header selection — then merges multi-row headers with last-row-wins semantics and backward-fills labels for columns the primary header row left blank.

Type inference classifies per-cell (numeric with currency/percent/parenthesized-negative cleaning; temporal with ISO 8601 canonicalization; partial dates like `1871.10`), then runs refinement passes over the census: integer columns labeled "date" whose values all parse as valid YYYYMMDD get retyped; datasets whose "header" row parses cleanly as data get demoted to headerless with synthetic column names.

After the Parquet is written, a statistics pass computes per-column count, distinct count, min/max/mean/standard deviation, quartiles, skewness, a 20-bin histogram, and Tukey-fence outliers — one DuckDB pass per column, persisted once. The dataset's Distributions and Quality tabs render from those precomputed rows in milliseconds; in v1 the same tabs recomputed on every view and took 30–90 seconds on large datasets.

Reshaping

Nineteen transform tools produce derivative datasets without leaving the browser: transpose, difference, lag/lead, rolling aggregates, time-series resampling, filtering, normalization (z-score and min-max), percent change, cumulative sum, four flavors of missing-value fill including linear interpolation, deduplication, group aggregation, pivot, rank, binning, statistical outlier removal, merge/join, append/stack, and a row-level clean tool. Each is a DuckDB SQL plan streamed into a fresh Parquet — the output is a real dataset, immediately usable in experiments.

A twentieth tool works differently: **computed columns** add expression-defined columns to a dataset *in place*. The owner writes an Excel-style expression over the existing columns — `ROUND([Price] * 1.08, 2)`, `IF([Score] >= 60, 'Pass', 'Fail')`, `DATEDIFF('day', [Listed], TODAY())` — in an editor with a column picker, a categorized library of ~40 functions, and a validate-and-preview pass

against the first rows of real data. The expression compiles through a typed parser to a DuckDB projection (user text never reaches SQL directly, so the surface is injection-proof by construction), the dataset's Parquet is rewritten with the extra column under a fresh storage key, and the reference swaps atomically — a failed computation can never damage the dataset. Messy real-world values degrade to nulls rather than errors, and a conditional like `IF(condition, 1, 0)` deliberately lands as a numeric column so it can participate in correlation runs.

The Experiment Engine

An Experiment pairs two datasets and exhaustively correlates them: every numeric column on X against every numeric column on Y. The engine's job is to make "test everything" cheap, deterministic, and statistically honest.

Joining

Two datasets rarely share row identity, so the engine supports four join semantics:

- **Row sequence** — positional pairing (X row *n* with Y row *n*) for pre-aligned data, with numbering assigned before null filtering so a missing value in row 3 doesn't shift every subsequent pair.
- **Shared key** — exact match on a designated key column (country codes, product IDs, exact timestamps), with duplicate keys aggregated per a per-column aggregate function (average by default; sum, min, max, first, last, count available).
- **Time series** — the workhorse. Both sides snap their timestamps to a common epoch grid controlled by one tolerance knob (hourly, daily, weekly, monthly, yearly buckets), so daily X data joins hourly Y data by averaging Y within each day. The bucket expression is deliberately timezone-stable and shared verbatim between the join and the drilldown query — a lesson learned when a timezone-dependent cast produced charts that worked and drilldowns that silently returned nothing. The key parser reads the date shapes official statistics actually publish — `1960M01` monthly codes, quarters, ISO weeks, bare years, `1871.01` — rather than only ISO dates; before v2.9.0, those matched nothing,

silently.
- **Geography** (v2.7.0) — datasets pair on shared place: the most specific geographic column type both carry (ZIP code, with census ZCTAs treated as the same geography; city; FIPS code; county; state; province; country; continent), or a latitude/longitude pair matched to about eleven meters. Keys normalize the forgiving way real data needs — case-folded text, canonical five-digit ZIPs, FIPS codes with their leading zeros restored — and because many rows usually share one place, each column aggregates per location by its aggregate function.

Sampling, deterministically

Experiments can run on a percentage sample. Sampling is hash-based on the join key — both sides keep a key if `hash(key) mod 100` falls under the sample percentage — so the surviving pairs always line up, and re-running the same experiment yields the *same* sample. Published discoveries are reproducible artifacts, not lottery draws.

The statistics

For each column pair, one analytical query computes:

- **Pearson's r** over the joined, null-and-NaN-filtered pairs, via DuckDB's numerically stable `CORR` aggregate.
- **Spearman's ρ** as Pearson over ranks — computed in the same pass with window-function ranking.
- **p-value** via a two-tailed Student-t test: $t = r \cdot \sqrt{[(n-2)/(1-r^2)]}$ with n−2 degrees of freedom.
- **95% confidence interval** via the Fisher z-transform: $z = \operatorname{atanh}(r)$, standard error $1/\sqrt{(n-3)}$, back through $\tanh$.

Pairs with fewer than three surviving points, zero variance on either side, or |r| below the experiment's threshold don't become discoveries. Two additional gates keep the catalog honest: an **identity gate** drops pairs whose correlation rounds to exactly 1.0 (a column against itself in disguise), and a **duplicate gate** skips column pairs the user has already mined in other experiments — before any computation happens — so re-running a growing cross-matrix costs only the *new* pairs.

Each surviving discovery gets a compact chart payload: up to 5,000

joined points, selected by deterministic hash ordering (same input, same sample), written as a small Parquet of its own. The full surviving pair count *n* is what the statistics report; the 5,000-point cap is purely a rendering budget.

Throughput on the hot path went through a deliberate optimization arc — batched inserts, payload-format changes, and finally the v2 columnar rebuild — moving from 3 pairs/second to a sustained 14–15 with peaks over 20. An "everything against everything" run across dozens of datasets is a coffee break, not an overnight job.

Living Datasets

A dataset ingested once is a snapshot, and the most valuable public data — economic series, prices, counts — keeps moving. Since v2.9.0 (October 2026), a dataset whose source is a URL can keep itself current. The owner picks an interval, from daily to quarterly — while creating the dataset or any time after — and a background worker re-checks the source on schedule, cheapest test first:

1. **Ask the server.** A conditional request (`If-None-Match` / `If-Modified-Since`): an unchanged file costs one round trip and nothing else.
2. **Compare the bytes.** A SHA-256 of the download against the previous one.
3. **Compare the rows.** Parse into a temporary Parquet and compare the row count and an order-independent row checksum against the live file, so a re-published file carrying identical data still counts as unchanged.

Only a real change is swapped in, and the swap is all-or-nothing: one conditional update that succeeds only if the dataset still points at the file the refresh compared against, so a concurrent re-ingest or delete always wins and nothing half-applies. If the process dies between the swap and the follow-up work, the dataset carries enough state to resume exactly where it stopped. Columns must still line up — a renamed header passes; a moved or missing column stops the refresh rather than silently pairing the wrong data — and a source that keeps failing turns its own refresh off and tells the owner. A refresh is billed like a manual re-ingest (the new file charged, the replaced one refunded), and every token it

moves is a line item in a per-run log that reconciles exactly with the ledger.

Experiments follow their data. One set to **auto-regenerate** recomputes every existing discovery *in place* when a dataset it uses refreshes: the same permanent URL, comments, ratings and written analysis, with new statistics — and the previous r kept, so the page can say "Updated · r was 0.42". Every pair is computed before anything is written, then the discoveries and the billing commit together in one transaction. Experiments that don't opt in are marked stale instead, exactly as after a manual re-ingest.

And what changed is visible. Refreshed content moves its sitemap date, is re-announced to search engines, and joins a daily **What's New** band on the home feed and a `/whats-new` timeline — newest first, grouped by day — whose shared link previews as a composite of the latest charts and tables. An administrator utility can put a whole corpus on per-series schedules at once: live daily and weekly series weekly, other live series every two weeks, everything else monthly.

From Correlation Toward Causation

The platform's second brand promise is *causation analysis*, and it's handled with statistical honesty: correlation is never presented as causation, but the tools to probe the question are one click away.

Granger causality asks the falsifiable version of the question: do past values of X help predict future values of Y beyond what Y's own history already predicts? The implementation is the full classical pipeline:

1. **Stationarity check** on both series via an Augmented Dickey-Fuller test; non-stationary series are differenced (up to twice) before testing.
2. **Optimal lag selection** by minimizing the Bayesian Information Criterion across candidate lags.
3. **F-test** comparing the restricted model (Y on its own lags) against the unrestricted model (Y on its own lags plus X's lags): `F = [(RSS_r − RSS_u)/k] / [RSS_u/(n−2k−1)]`.
4. **Both directions.** X→Y and Y→X are tested separately; a discovery reports both F-statistics, both p-values, the optimal lag,

and a four-state direction summary (none / X→Y / Y→X / bidirectional).

The UI language is careful: Granger causality is *predictive* causality. It cannot rule out a third variable driving both series — and the platform's own AI analyses say so explicitly when the data warrants it.

Three lighter-weight instruments complete the causality toolkit:

- **Lag analysis** slides one series against the other and re-computes r at every offset, revealing lead/lag structure visually.
- **Rolling correlation** sweeps a window across the joined series to expose whether the headline r is stable, decaying, or hiding a sign flip.
- **Divergence detection** compares |Spearman| against |Pearson| per discovery. When Spearman is meaningfully higher, the relationship is monotone but non-linear (Pearson under-reports curves); when Pearson is meaningfully higher, outliers are propping it up. The platform flags both cases automatically, right on the discovery list.

Regression rounds it out: linear, quadratic, cubic, logarithmic, and exponential fits via ordinary least squares, each with R^2 and a 95% prediction interval band drawn on the chart, plus a residual-plot mode for diagnosing what the chosen model misses. A model-comparison panel fits all families at once and ranks them.

Ten Ways to See a Relationship

Every discovery renders through a purpose-built visualization layer:

View	What it shows
Scatterplot	The relationship itself, with density-bucketed dots and regression overlay
Line graph	Both series against row order, dual-axis, density-aware alpha
Residual plot	Distance from the fitted curve — fit diagnostics at a glance

Trajectory	Grouped series tracing arcs through (x, y) space over time
Heatmap	The full r matrix across an experiment's column pairs
Sparkline grid	Every discovery in an experiment as a wall of mini-charts
Bubble chart	r vs sample size, sized by non-linearity
Divergence chart	Pearson vs Spearman per discovery, against the agreement line
Lag chart	Correlation at every time offset
Correlation network	A force-directed graph of which columns move together

The primary chart engine is canvas-based and double-buffered (mode switches never flash), with density handling designed for real data: scatter points collapse into buckets with logarithmically-scaled radii, and dense line renders modulate stroke alpha so 30,000 segments read as density rather than a solid band. A quiet guard computes lag-1 autocorrelation on both series and warns when a line graph would be misleading because row order is noise.

Clicking any datapoint drills down to the **actual source rows** from both datasets that produced it — bucket-aware, so a daily time-series point returns every X row and every Y row from that day, streamed to the browser as they're read. The chain of custody from chart to raw data is never more than one click.

Every public discovery also gets a server-rendered thumbnail (the same bucketing math, rendered headlessly), which is what makes feeds, portfolios, search results, and social-media unfurls visually rich. Datasets with geographic columns get a map of their own: a representative sample drawn from across every row, with pins clustered by zoom level.

The AI Layer

AI shows up in four places, each priced and audited:

Search (described above): three providers in parallel, results verified before presentation.

Analysis: any dataset, experiment, discovery, or portfolio can be analyzed by Claude, Gemini, or Grok. The model receives the real statistics — coefficients, p-values, confidence intervals, Granger results, column semantics — and produces a structured narrative: overall relationship, strength and direction, patterns and outliers, confounding factors, actionable insights. The narrative is pinned to the entity, rendered as Markdown, editable by the owner, and re-generable on demand. Analyses are honest by construction: given a modest r with no Granger signal, the write-up says co-movement from shared macro factors, not causation.

Corrie, the in-app assistant, is a retrieval-augmented chatbot over two corpora: the product documentation, and every public entity on the platform. The pipeline is classical RAG done carefully:

- **Embeddings** are dual-vendor: OpenAI's `text-embedding-3-large` (truncated to 1536 dimensions) as primary, Gemini's embedding model as automatic fallback on transient failures — same dimensionality, so the vector index doesn't care which vendor produced a given row.
- **Indexing is lifecycle-driven.** Publishing, unpublishing, deleting, or re-analyzing an entity enqueues a durable index command; a background worker batches embeddings (hundreds per API call), retries with exponential backoff, and never gives up on a transient failure. The corpus tracks the platform in near-real-time.
- **Retrieval** is pgvector cosine similarity — the top three documentation chunks plus the top ten entity chunks — with a visibility filter so private content never leaks into another user's chat.
- **Synthesis** runs on Claude Haiku for speed and cost. When the retrieved context doesn't cover the question, Haiku is told to say so with a sentinel token rather than guess, and the question escalates to Sonnet. Both models come from an operator-maintained catalog, so moving to a newer model is a catalog edit, not a deploy. Every answer carries **citations** that link to the underlying discoveries, and every query writes an audit row with token counts and cost.

Ask Corrie how the S&P 500 relates to oil prices and the answer is grouped coefficients across time periods, each citation-linked — with the caveats stated, like Granger tests showing no significant predictive direction.

Classification: a Haiku-based classifier assigns every public experiment and dataset to a curated two-level topic taxonomy by reading the analysis text the platform already generated. Discoveries inherit their parent experiment's topics — one column pair shares its parent's subject — so the entire 50,000-discovery corpus is topic-organized at the cost of classifying only experiments and datasets.

Open by Design

A public corpus is only valuable if it can be found. A React SPA is invisible to anything that doesn't execute JavaScript — which includes search-engine first passes, LLM crawlers, and social-media unfurlers. The discoverability layer (v2.4.0, late June 2026) fixed that end to end:

- **Server-rendered entity shells.** Every public dataset, experiment, discovery, and portfolio URL serves real content before JavaScript runs: a proper title and description, Open Graph and Twitter Card tags pointing at the entity's chart thumbnail, a readable article body (the analysis narrative, the coefficient block, the column roster) for non-executing crawlers, and schema.org JSON-LD — `Dataset` for datasets with a real downloadable distribution, `Article` for analytical entities, breadcrumbs throughout. Human visitors get the SPA exactly as before; the server-rendered layer exists for the first, non-JS fetch. The rewrite adds nothing measurable to latency because it templates data the API already loaded.
- **Topic hubs** turn the taxonomy into public, crawlable index pages — an internal-linking mesh connecting every entity to its subject neighbors, which is the single biggest lever on how deeply crawlers explore a site.
- **Freshness signals**: sitemaps with true per-type modification dates, an image-sitemap extension advertising every discovery's chart to image search, and IndexNow pings fired asynchronously on every publish so participating engines recrawl within minutes.

- **Freshness you can see.** When a dataset refreshes from its source, it and the discoveries recomputed on it move their sitemap dates, are re-announced through IndexNow, and are listed under "Recently refreshed" in `llms-full.txt`. Their server-rendered pages say when they were refreshed or recomputed (a discovery also shows its previous r), with `temporalCoverage` in the structured data. The `/whats-new` page server-renders its own list of the newest updates, and a composite share card built from their charts and tables; `/whats-new.atom` carries the same timeline as an Atom feed, listed in `robots.txt` (search engines read a feed as a sitemap of what's newest) and in `llms.txt` (agents can poll it rather than re-crawl). IndexNow notifications are batched — one request every couple of minutes, each URL at most hourly — since a refresh day changes thousands of pages and a stream of small requests is what engines throttle.
- **Open data, three ways.** The full public corpus ships as `llms-full.txt` (the whole corpus as one Markdown document, following the emerging convention for LLM consumption); as a streamed NDJSON bulk dump where every record is self-describing with license and attribution embedded; and as per-dataset anonymous CSV exports referenced from each dataset's JSON-LD — which makes every public dataset, and the corpus itself, discoverable in Google Dataset Search. License: **CC BY 4.0**, attribution to the content's creator.

The bet is simple: analytical content with real statistics and honest write-ups is exactly the kind of material search engines and AI answer engines want to cite. Making it maximally machine-readable is marketing that compounds.

The Token Economy

The business model is engineered to keep the free tier genuinely free and the paid path honest:

- **Free to join. No credit card.** Every account gets **10 GB of storage** with no recurring charge, and new registrations receive a starting token grant.
- **Tokens are the metered unit.** Ingestion, correlation runs, transforms, AI search, AI analysis, and chatbot queries each carry

a posted token price (ingestion and correlation scale with data size; AI operations are flat per call). Token packs range from $5 to $200 with a deliberately tight volume discount, and **tokens never expire**. There is no subscription requirement; one optional storage subscription tier exists for users who outgrow 10 GB.
- **Deleting content earns credit back** — a large fraction of the size-based portion of the original ingestion cost returns to the balance, so experimentation isn't punished.

Under the hood, the accounting is bank-grade in miniature. Every token movement writes a ledger row whose running balance comes from the same atomic database operation that moved the balance — a lost-update race in an early read-modify-write implementation produced real drift in production, and the fix was to make drift structurally impossible rather than merely unlikely. Stripe integration is idempotent at every entry point (webhooks retry aggressively for days; every handler dedupes on event identity), invoices are generated as PDFs with monotonic numbering and snapshotted billing details, and a weekly reconciliation sweep cross-checks stored balances against the ledger and storage counters against actual bytes.

The unit economics close at small scale by design. Infrastructure is two VPS instances, a CDN, and per-byte object storage with zero egress — roughly $300 a month all-in, independent of corpus size in any way that matters. The only cost line that scales with engagement is AI API spend, which is exactly the line token pricing covers. A few dozen actively-engaged users make the platform self-funding; everything above that is margin. The free tier can stay free indefinitely because storage is the cheap part and idle users cost approximately nothing.

Running It

The operational posture is *recovery-first*: assume the process can die at any moment, and make every in-flight operation either resumable or safely restartable.

- Dataset and experiment pipelines run on **single-column state machines** with deterministic startup recovery — a crash mid-ingest resumes or requeues based on exactly what survived (partial file on disk, upload still staged, download resumable by

byte offset).
- Long-running remote downloads are **durable across restarts**, with heartbeats, partial-state detection, and byte-range resume.
- Background work runs under **leader election** so a blue/green deploy can't double-charge or double-process.
- In August the database moved to a right-sized server by restoring it from its off-site backups, with zero data loss.
- Every external integration assumes retries: **idempotent webhooks**, embedding-queue commands that back off exponentially and never give up, object-storage uploads that rebuild their TLS client when the connection pool itself is the problem (a real production failure mode: a poisoned pooled socket failing identically on every retry until the pool was forcibly replaced).

Trust boundaries got the same treatment in October. Every read and every action goes through **one visibility rule** — the owner, the public, or a workgroup's members — applied identically to pages, rows, exports, charts, drilldowns, comments, tags and status polls. Owner-only actions, like re-running an experiment or analyzing it, are owner-only everywhere. And every outbound fetch of a user-supplied URL (downloads, refreshes, link previews, thumbnail fetches) checks each address it connects to, redirects included, so the server can't be pointed at itself. Both the refresh pipeline and those rules are verified by end-to-end harnesses that boot the real application against a throwaway database and a fake object store; the security harness sends real requests as six kinds of user and runs a network canary that must never see a connection.

A 30-day operational snapshot from early summer 2026, under live ad-driven traffic:

Metric	Trailing 30 days
Requests	~1,000,000
Unique visitors (IPs)	~45,000
Average request duration	79 ms
Server errors (5xx)	5 total
Uptime	100.00%

Five server errors in a million requests, at 79 milliseconds average latency, on two modest servers — with the crawler-facing SSR layer in the request path. That table is the whole architectural argument compressed: the system that fell over at ten concurrent users now doesn't notice a million-request month.

The Timeline

The compressed history, for the engineering-curious:

- **Dec 29, 2025** — commit zero. Three-layer .NET solution + React SPA. The layout never changes.
- **January–February** — dataset ingest plumbing; Discoveries and Experiments become real entities.
- **March** — public IDs, the token economy, AI-powered dataset search (Claude first, then Gemini), remote downloads, soft-delete.
- **Early April** — Grok joins as the third search provider; ingestion re-architected; state machines replace boolean-flag soup; the token ledger lands.
- **Mid-April** — the visualization wave (heatmaps, divergence, bubble, lag, rolling, network) and the social layer (portfolios, posts, messaging, forums, workgroups) in a single furious fortnight.
- **Late April** — migration from SQL Server to PostgreSQL; database sharding built out; Granger causality ships; Stripe lands end-to-end.
- **Early May** — the performance arc (3 → 15 pairs/sec), the RAG chatbot built and shipped in two days, resilience hardening across the wizard and download paths.
- **May 17** — **the v2 Lakehouse migration.** Sharding deleted, Parquet + DuckDB in, ~5,000 lines of C# removed, frontend untouched. The defining architectural event of the project.
- **Late May** — token-flow hardening (an external audit closed in five sequential phases), content moderation, activity limits, security audit.
- **June 1** — public launch. Ad campaigns, funnel instrumentation, and the honest lessons of early go-to-market (the first campaign that actually converted was desktop-only with an exclude-list geo strategy — measured bursts beat always-on spend).

- **Late June (v2.4.0)** — the discoverability release: crawler SSR, structured data, topic ontology, IndexNow, open-data surfaces.
- **Early July (v2.5.0)** — the crawler: systematic, robots-respecting dataset harvesting at portal scale, feeding the corpus that now exceeds 55,000 discoveries.
- **Mid-July (v2.6.0–v2.7.0)** — the layered home feed with daily AI-curated trending bands, and the GeoType join: datasets pairing on shared geography (ZIP/ZCTA, city, county/FIPS, state, country, or coordinates).
- **Late July (v2.8.0)** — computed columns: expression-defined columns materialized onto existing datasets in place, with an injection-proof expression compiler over DuckDB.
- **August–September** — an operational stretch: the database moved to a right-sized server by restore-from-backup with zero data loss; the dataset map learned to sample whole datasets and cluster its pins; repeat-visitor analytics; traffic reporting corrected at the source.
- **Early October (v2.9.0)** — living datasets: scheduled refresh from the source, experiments recomputed in place, a What's New timeline with its own share card — and a platform-wide security pass, verified by end-to-end harnesses.

Nine months, ~850 commits, one engineer.

Lessons That Survived Contact

Every one of these was learned in production, not in a design review:

1. **One state column beats four boolean flags.** Every queue bug in the project's first four months traced back to flag combinations no one had enumerated. A single authoritative state with one writer per transition made startup recovery deterministic and killed the bug class.
2. **Make financial drift structurally impossible.** Balances move only through atomic database operations that return the new value; the audit ledger records that returned value, not a separately-computed one. Reconciliation then verifies invariants instead of repairing them.
3. **Delete the clever thing when the simple thing proves more**

durable. Sharding, per-user content databases, cross-database replication schemes, four-flag state tracking, in-database bulk row storage — each was replaced by something with fewer moving parts, and every replacement paid back faster than projected.

4. **Match the storage engine to the read pattern.** The v1 failure wasn't a bug; it was a row store serving a columnar workload. No amount of tuning crosses that gap.
5. **Build the same expression once.** The join-bucket bug — charts working while drilldowns returned nothing — happened because two code paths computed "the same" key two subtly different ways. The fix wasn't the timezone handling; it was making one function the only source of that expression.
6. **Idempotency is not optional at the edges.** Payment webhooks, embedding queues, download resumes, crawler frontiers — anything that talks to the outside world will be retried, replayed, or duplicated eventually.
7. **Real-world files are the adversary.** Six iterations of header detection, each falsified by one government CSV. Parsers earn trust one hostile file at a time.
8. **Replace live data with a compare-and-swap.** The refresh pipeline's swap is one conditional update that succeeds only if nothing else changed the dataset since the refresh read it. Anything subtler — a check-then-write, tracked changes saved halfway through a transaction — eventually loses a race to a concurrent re-ingest, or commits half of a retried attempt.
9. **A deploy ships everything that's waiting.** A change committed for one feature carries its migration into the next, unrelated publish. Deploy notes list every pending migration, not just the one for the change at hand.
10. **A solo build's edge is loop time.** Not being smarter on paper — closing the distance between "this broke in production" and "this is fixed and can't break that way again" in hours instead of sprints.

Where It Goes From Here

The platform is feature-complete for its core personas — analysts, researchers, students, and the incurably curious — and default-alive on the economics. The corpus compounds through the crawler, and now

keeps itself current through scheduled refresh. The open-data surfaces are in the wild, accumulating citations. Horizontal scale-out, when traffic demands it, is architecturally trivial: the app server holds no shared mutable state, and both Postgres and R2 are already shared substrates.

The premise from the top of this document was that one engineer with the right primitives could build and operate a system that historically took a team. The evidence is the system you're reading about — and the corpus it built.

See for yourself: browse the live corpus, explore the open data, check pricing (free to join, no credit card required), or just create an account and upload something. Your data is hiding something. Find it this afternoon.

© 2026 Correlation Studio LLC. The public corpus is licensed CC BY 4.0. Questions about the platform or this document: via the in-app contact form at correlationstudio.com.