Kodiak v1

Kodiak v1 is the MVP iteration of the OLLIE Project: the *Open-source Ledger Linking Information to Evidence*. Kodiak exists to *find, map, correlate, archive, and display geopolitical data related to corporate responsibility*. The project was born out of a growing problem: our public archives are *increasingly* incomplete. Information disappears quietly, often for reasons that have little to do with accuracy and much to do with optics.

No major archival service currently operates with canary notices or transparent deletion records. In some cases, archives have even refused to preserve material because it was deemed "*disagreeable*." When the record of the past can be selectively trimmed, trust erodes. Archiving the internet is no longer just a technical act, it's an act of accountability.

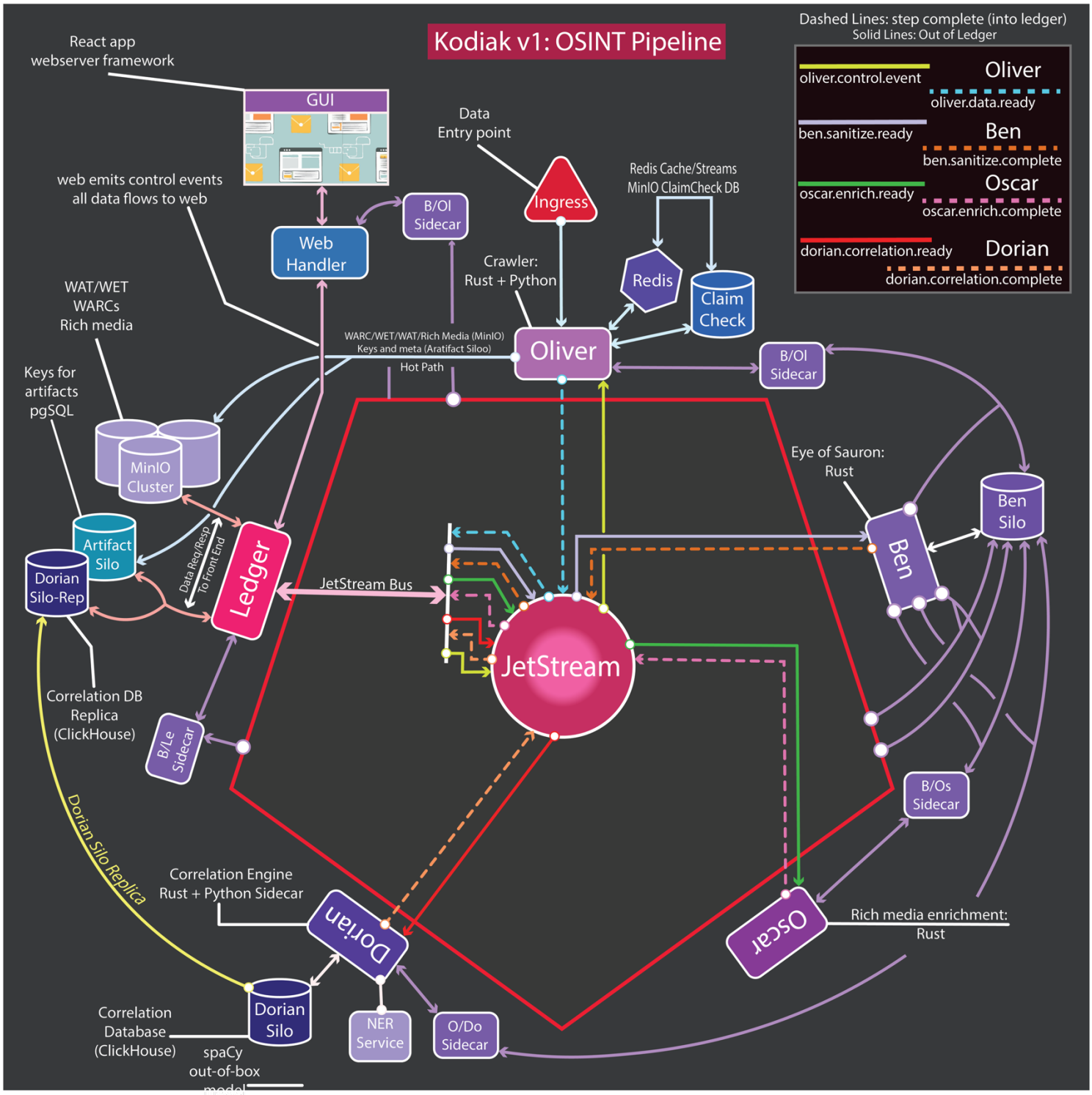
Modern news and information systems are driven by narrative rather than documentation. Stories are published, revised, and quietly removed. Platform moderation hides or throttles data that challenges prevailing sentiment. The result is a kind of informational fugue state: the moment something vanishes from view; collective memory begins to dissolve.

Archiving is only the first layer of the *Kodiak v1*. Its deeper purpose is correlation, to connect disparate fragments of information and restore lost context. Data in isolation is noise; correlation turns it into structure. Through its pipeline, Kodiak extracts and analyzes entities, establishes statistical relationships, and reveals patterns invisible from ground level.

The system is designed for NGOs, journalists, and independent researchers who need stable access to geopolitical and corporate data. OLLIE Project **does not accept** government, corporate, or institutional funding, nor does it grant access to those entities. This separation is not a gesture of hostility but one of integrity. In an arena shaped by perception, even the appearance of partisanship compromises trust.

The OLLIE Project is not built to sell narratives; it's built to preserve evidence. Its mission is to give voice to what is suppressed and to ensure that crucial events are not erased by convenience or neglect. Memory, once lost, cannot be reconstructed. Kodiak v1 exists to prevent that loss.

Kodiak v1 Overview



Design Ethos

The **Kodiak v1** comprises a number of interconnected subsystems. Many of these could, in theory, be implemented entirely in Python, but that would introduce inefficiencies and unnecessary complexity. Instead, the design ethos favors a **hybrid architecture**: lightweight Python services running in isolated containers, orchestrated by a Rust core.

This approach provides several advantages:

- Enables precise and reliable system monitoring.
- Minimizes the surface area for memory-related bugs or exploits.
- Ensures Python services never interact directly with the filesystem.
- Facilitates robust retry queues and message persistence.
- Allows failing services to restart independently, without cascading failures.
- Assigns one process per machine learning model for controlled isolation.
- Allows for rapid worker replacement and scaling.
- Leverages Rust's macro system for dynamic orchestration and extensibility.

The central goal is to **minimize Python's footprint** while maintaining flexibility. All orchestration between Rust and Python occurs through **Protocol Buffers (protobuf)**, with immutable contracts that ensure compatibility and stability. Each subsystem operates as a **job-driven unit**, defined through JSON-based configuration.

Extensibility and modularity are foundational principles. The architecture embraces a "do more with less" mindset: decorators, middleware, and macros are not layered for ornamentation, but to reduce code bloat while maximizing functionality and composability.

Data flow moves downward from **Oliver** through to **Dorian** in this iteration. Later designs would aim to establish a fully automated feedback loop designed to become **self-regulating** once operational. Human oversight would exist, but primarily focused on ethics, fail-safes, and anomaly monitoring.

The system would heavily rely on the **Historical Silo** to process data targets before passing refined analysis for targets, In later iterations orchestration back to **Oliver** for targeted harvesting would be done by an "ingress orchestrator" named **Cooper**. The design is **recursive, ethical, and adaptive**. It is a system that learns, validates, and self-corrects without human micromanagement.

System Parts

Oliver

Oliver is the primary crawler subsystem of **Kodiak v1**. It is responsible for targeted data acquisition: discovering, validating, fetching, and staging web resources for downstream processing. Oliver is designed for extensibility and modularity so the crawler can scale from low-effort harvests to resource-intensive collection tasks without destabilizing the broader system.

Responsibilities

- Target discovery and reconnaissance
- Adaptive retrieval through multiple fetch strategies
- Proxyed, streamed capture of HTTP(S) traffic into a canonical sink
- Immediate persistence using the claim-check pattern (MinIO + WARC)
- Generation of derivative WAT/WET objects for downstream NLP and indexing
- Robust retry, backoff, and throttling controls to maximize longevity and reduce ban risk

---

Recon & Targeting

Reconnaissance precedes automated harvesting and serves two purposes: (1) locate the relevant data with minimal waste, and (2) surface site defenses that determine retrieval strategy. Oliver emphasizes targeted crawling (contrasting with broad indexing crawlers) and combines automated reconnaissance with human verification for high-value targets until the Historical Silo becomes a reliable source of truth.

Key recon checks:

- Site structure and canonical entry points
- Presence and sophistication of anti-bot systems
- API availability and legal/ethical constraints

Anti-bot posture & multi-signal hygiene

Oliver treats TLS fingerprinting and similar signals as *one* input to a multi-factor decision engine. Defenses are designed to be holistic rather than TLS-only:

Signals we normalize and monitor:

- HTTP header consistency and semantics
- ALPN / HTTP/2 framing behavior
- JavaScript-observable runtime characteristics (when available)
- Request timing and rate patterns consistent with human usage
- IP / ASN reputation and session linkage

Operational mitigations:

- Fallback retrieval flows for blocked targets
- Logging/timestamping of fingerprint changes for fleet visibility
- Initial focus on “soft” data-rich targets while maturing the stack
- Legal and API-first retrieval where available

Tiered retrieval strategy

Oliver uses a tiered escalation model to match resource cost to target complexity:

Tier	Approach	Nickname	Purpose
1	reqwest	Handgun	Static pages, public APIs or weakly guarded information
2	Playwright+Stealth Plugins	Rifle	JS heavy pages, light bot defense, standard captcha solving
3	Crawl4AI/Nodriver	Ordinance	Invisible CAPTCHAs, heavy anti-bot defenses, flow control defenses, heavy obfuscation

Rationale: preserve expensive headless/stealth resources for targets that require them; run low-cost clients where possible to maximize longevity and reduce detection risk.

Retrieval pipeline & MiTM proxy

All retrievals are routed through a containerized MiTM proxy. The proxy provides: cross-service streaming of traffic, tracker blocking, uniform data capture, and a single TCP sink interface. Using one proxy simplifies scaling: new worker types (Playwright, Nodriver, Rust HTTP clients) attach to the same capture point and stream chunked data to the Rust TCP sink.

Benefits:

- Single source of truth for raw traffic
- Ability to sanitize or block trackers at capture time
- Container orchestration of proxy instances for horizontal scale
- Early data caps and flow controls to prevent self-DoS

Claim-check persistence & WARC derivatives

Oliver follows a claim-check pattern: raw payloads are stored immediately in MinIO, returning opaque keys which are then referenced by WARC writers. From the primary WARC we produce:

- **WAT** - structured metadata for each record (links, headers, digests)
- **WET** - extracted text for NLP and indexing

Rich media (PDF, audio, video) are stored as media objects with WAT metadata; full semantic extraction of those media is considered out of scope for initial harvesting and deferred to downstream workers.

Orchestration, macros, and job model

Orchestration is implemented in Rust and coordinates containerized Python workers via protobuf contracts. Key aspects:

- Immutable service contracts via Protocol Buffers
- Jobs are declaratively specified by JSON and executed by orchestrator-scheduled workers
- Attribute macros in Rust provide reusable templates for common concerns (heartbeats, health checks, instrumentation) and reduce boilerplate across job definitions
- Middleware and decorators enable composable pipelines without code duplication

This hybrid approach reduces the Python surface area while leveraging Python where necessary (e.g., Playwright agents, certain NLP prefilters).

Fleet state, caching, and throttling

Redis is used for both caching and queueing:

- **Cache:** avoid redundant crawls and store short-lived analytics/state used by **Oliver**
- **Queue:** Redis Queue implements a configurable “turnstile” - host-level throttles and prioritization that can be hot-swapped at runtime

The turnstile enforces host-specific backoff rules to reduce blacklist risk and respects site-level politeness constraints.

Jobs & configurability

Jobs are highly configurable JSON objects delivered by Cooper. Each job can specify:

- Target URL(s), crawl depth, and discovery rules
- Preferred retrieval tier and fallback strategies
- Rate limits, concurrency, and retries
- Post-capture processing hooks (WARC emission, WAT/WET generation, downstream enqueueing)

This configuration-driven model supports per-target specialization and rapid adaptation without changing worker code.

Operational note: ***Oliver*** aims for automation-first operation with minimal human intervention. Human actors remain responsible for high-risk decisions, ethical reviews, and exceptional access workflows. Monitoring, fail-safes, and audit trails are integrated to maintain a high ethical bar.

Lastly, Oliver runs jobs based on complex JSON fed to it by users, or when ready, Cooper. Targets can have very specific configurations used with that approach. There is no “one size fits all” approach. More so, it is a “universal remote” where there is never one thing that can’t be augmented with ease.

Ben

Operational Note:

Ben ensures that every subsystem in Kodiak,Oliver, Oscar, Dorian, and Ledger remains transparent, measurable, and self-correcting through unified telemetry and autonomous triage.

Ben is **the observability engine with teeth** - a distributed nervous system that sees, interprets, and reacts to every pulse of the Kodiak stack.
Its purpose is not just to watch, but to *understand*: to establish baselines, detect anomalies, and initiate triage before a human even notices something's wrong.

Ben is embedded everywhere - **Oliver, Oscar, Dorian, Ledger** - through compile-time **attribute macros** and shared **trait integrations**. This ensures that every subsystem emits consistent, structured telemetry that Ben can interpret in context. The result is a living system that is **observable, explainable, and self-regulating**.

Earlier iterations of Kodiak relied on single-point "sanitizers" to approve or quarantine data. That approach proved too brittle; a single lock can always be picked. Ben instead acts as a **lymphatic system**, circulating observability throughout the organism so that when data infection or behavioral anomalies arise, containment and response happen quickly.

Most organizations bolt observability on after the fact. Kodiak bakes it in at the genetic level.

Ethos

"As resilient as Rasputin, observant as God."

Guiding Principles

1. **Total Observability** – every subsystem emits structured, contextual telemetry; nothing is opaque.
2. **Defense in Depth** – all signals pass through multiple layers of validation, redaction, and policy before acceptance.
3. **Autonomous Reaction** – when a drift or spike occurs, Ben doesn't just report it; it responds within bounded authority.
4. **Explainability** – every alert, correction, and decision is reproducible, timestamped, and human-readable.

Machine Learning Core

Ben is **machine-learning-driven**. The scale and heterogeneity of telemetry make manual oversight impossible.
It maintains its own high-throughput **ClickHouse** database along its hot path, allowing for real-time baselines and long-term trend analysis.

Baselines are not static. As Kodiak evolves and adds new data sources, Ben must continually relearn what "normal" looks like. Each subsystem is first observed in an **isolation or test environment**, allowing Ben to characterize normal distributions and relationships. Once those baselines stabilize, the data source is promoted to production, carrying its learned parameters forward.

Drift Management

Drift is inevitable in complex systems - workloads shift, dependencies update, environments change.
Ben handles drift through a combination of **adaptive thresholding** and **scheduled re-calibration**:

- **Adaptive Thresholding:** short-term adjustments occur automatically using statistical detectors (EWMA, CUSUM) that adapt to slow changes in distributions.
- **Periodic Re-Calibration:** Ben periodically re-samples telemetry across all subsystems during low-load windows, comparing current behavior against historical baselines.

Deviations are analyzed using KL-divergence and mutual-information metrics to flag structural shifts.

- **Human-Readable Drift Reports:** before parameters are accepted as “the new normal,” Ben generates digestible reports for operator review - visual diffs of old vs. new baselines. This enforces transparency in self-adjustment.

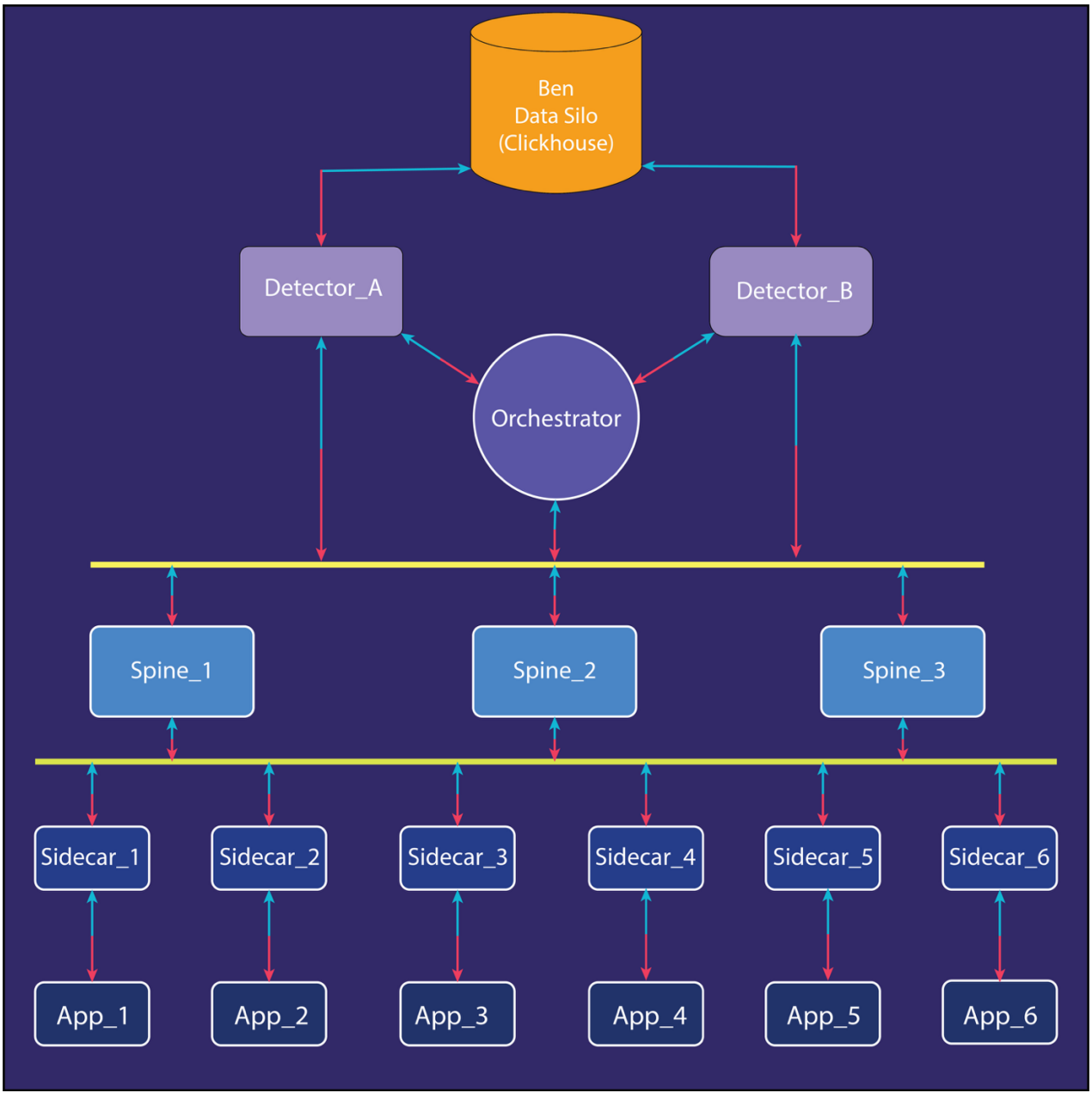
Operational Philosophy

Ben is designed for **bounded autonomy**. It can act without permission, but never without explanation.
If a subsystem fails, Ben isolates it; if data misbehaves, Ben quarantines it; if performance erodes, Ben triages it.
Its primary interface to human operators is narrative reporting - **plain-language summaries backed by hard metrics** - so humans can audit machine judgment without parsing graphs or logs.

Ben’s presence turns the Kodiak stack from a collection of services into a **self-aware organism**: resilient, introspective, and capable of defending its own integrity.

High-Level Architecture of Ben

Ben operates as a **central spine** fed by a constellation of **sidecars**.



Components

- **Sidecars:** local daemons (Rust) living beside Oliver, Dorian, Oscar, etc.
    - Handle ingestion, sanitization, WAL buffering, and local triage.
    - Talk to Ben over gRPC streaming via Unix Domain Sockets.
- **Ben Spine:** central ingestion and analysis service.
    - Prioritizes lanes (Alerts > Anomalies > Routine).
    - Writes sanitized, structured data to ClickHouse.
    - Maintains detectors (EWMA/CUSUM) and streaming policies.
- Ben Orchestrator: central ingestion and analysis service.
    - reconciles desired versus actual state across spines and sidecars, scales resources, rotates keys, and heals the fleet without human intervention.
    - The system's conscience as much as its scheduler, ensuring automation neverdrifts into opacity.
    - re-calibrating baselines, spawning new spines, or re-enrolling sidecars: turning observation into resilience
- **ClickHouse:** authoritative storage for raw and derived facts.
- **Ledger (Postgres/ClickHouse small tables):** holds status beacons and control-state metadata.
- **Control Plane:** Ben issues commands back to sidecars (sampling adjustments, throttles, quarantines).

Sidecars are like military officers working with units. They give the orders should something arise, they also report status back to the headquarters. This is done to offload much complexity from Ben that would inherently make the codebase bloated and make it brittle. Sidecars also allow for different implementations that can be finetuned to the subsystem it interacts with.

Data Flow (Upstream)

1. **Emission:** subsystem generates signals (jobs, IO, artifacts, anomalies).
2. **Sidecar ingestion:** sanitizes, batches, compresses (zstd), and WALs.
3. **Streaming:** sidecar streams to Ben Spine over **gRPC+UDS**, using **credit-based flow control** for backpressure.
4. **Spine classification:** assigns lane priority, runs detectors, and persists events to ClickHouse.
5. **Incidents:** anomalies promote to *Incident* objects (structured, explainable).

Control Flow (Downstream)

Ben communicates back to sidecars via the same persistent stream.

Commands

Kind	Purpose
`set_mode`	Flip sidecar between OK / DEGRADED / BROWNOUT
`set_sampling`	Adjust telemetry sampling rate
`throttle_job`	Slow or pause a runaway job
`quarantine_artifact`	Denylist an input/output hash
`fetch_logs`	Request recent local logs for correlation

Each command is **signed**, **budgeted**, and **idempotent**, carrying a clear **deadline** and **explain** payload.

Acknowledgements

Sidecars respond with `CommandAck` (accepted/in_progress/succeeded/failed), ensuring full auditability and reproducibility.

Internal Systems

Ben's architecture mirrors Oliver's macro-first design. Attribute macros and registries define detectors, playbooks, and redactors declaratively, enabling safe, hot-swappable extensions without modifying the core. This ensures parity in extensibility across both observability and orchestration layers.

Ingest Spine

- gRPC bidirectional streams (UDS).
- Credit-based flow control: Ben tells producers how many batches to send.
- Bounded lanes with priority-based shedding (routine last, alerts never).
- Write-ahead logs (WAL) for crash recovery.

Detectors

Fast online algorithms detect drifts and anomalies in real time.

- **EWMA (Exponentially Weighted Moving Average)** for baselines.
- **CUSUM (Cumulative Sum)** for change detection.
- **Robust stats** for outliers, count-min sketches for rarity tracking.

Each detector emits an `Incident` with a full explanation:

```
{
  kind: "latency_spike",
  observed_ms: 9800,
  baseline_ms: 1200,
  method: "cusum(v=2.5,h=8)",
  explain: "CUSUM crossed threshold after 6.5σ sustained deviation"
}
```

Detectors in Ben are built as pluggable, hot-swappable modules: each implements a tiny Detector trait (ingest → optional incident) and is registered with a central *DetectorRegistry*. At runtime, the spine loads a pipeline config (TOML/YAML) that maps topics/metrics to ordered chains of detectors—e.g., latency_ms → [ewma{…},cusum{…}, outlier{…}] so adding the N-th detector is just shipping a crate and naming in config. Pipelines can be hot-reloaded without restarts; unchanged detectors keep state; new ones start warm from a lookback replay.

We enforce bounded autonomy with per-detector CPU/memory budgets, cooldowns, and backpressure; every emitting incident includes an explain block (params hash, thresholds, state hints) for auditability. This design gives you an arbitrary number of detectors, first party or third-party. Composed per metric, per tenant, or per service, without touching the spine's core.

Orchestrator

- The Nervous System's Reflex Arc
    - The orchestrator is Ben's autonomic layer, it doesn't analyze data like detectors keeps the entire organism alive. Reconciling desired versus actual state across and sidecars, scales resources, rotates keys, and heals the fleet without human intervention.
- Source of Bounded Authority
    - It embodies Kodiak's principle of bounded autonomy: the orchestrator can act: it spawns, throttles, quarantines, and every change is signed, logged, and explainable. It's the system's conscience as much as its scheduler, ensuring automation never drifts into opacity.
- The Conductor of Adaptation
- Working in-between the detectors, it converts insight into action. When detect mark anomalies or drift, the orchestrator responds: re-calibrating baselines, spawning new spines, or re-enrolling sidecars, turning observation into resilient

Storage Model

ClickHouse tables (simplified):

```
CREATE TABLE ben_raw_events (
  ts DateTime64(6, 'UTC'),
  host String,
  proc String,
  run_id UInt64,
  topic LowCardinality(String),
  payload_json JSON,
  ver UInt16
)
ENGINE = MergeTree
PARTITION BY toDate(ts)
ORDER BY (ts, host, proc, run_id, topic);
```

Materialized views provide rollups (p50/p95/p99 latency, counts, error rates).

Sidecar Responsibilities

Function	Description
Ingest Adapter	Converts subsystem logs to `Sig` structs.

Function	Description
Sanitizer	Redacts and tags sensitive fields.
WAL Buffer	Durably stores events when Ben is unavailable.
Transport Client	Manages stream, credits, reconnection.
Executor	Runs Ben's commands within budgets and deadlines.
Local State (RocksDB)	Stores seq numbers, incidents, and last configs.

Sidecars act as autonomous field agents with standing orders and local override capability.

Modes and Resilience

Operational Modes

Mode	Behavior
OK	Normal telemetry and command execution.
DEGRADED	Partial sampling; WALs noncritical data.
BROWNOUT	Summary-only signals; alerts still flow.
DOWN	WAL-only; replay on reconnection.

Resilience Patterns

- Atomic health beacon /run/ben/status.json (local + Ledger).
- WAL replay and snapshot recovery.
- Credit flow control for built-in backpressure.
- Supervisors for self-healing tasks.
- Graceful degradation ladder (never silent failure).

Security and Ethics

- **Redaction manifests:** every field filtered via declarative selectors.

- **Explainability:** each alert and command carries an “explain” object.
- **Replayability:** all incidents are reproducible from raw facts.
- **Kill switches:** emergency quiet mode per subsystem.
- **Auditing:** cryptographic signatures on commands; immutable command logs.

Future Expansion

- **ML Cortex:**
  - Adaptive threshold tuning (per job, per time window).
  - Contextual bandits for choosing playbooks.
  - Drift detection for data poisoning defense.
- **Policy-as-Code:** declarative control rules in TOML/YAML with signature verification.
- **Ben Federation:** multiple Ben instances sharing incident summaries through JetStream for horizontal scaling.
- **Web Integration:** real-time cluster map (status beacons + incidents feed).

Summary

Ben transforms Kodiak from a collection of data processors into a **living, self-observing organism**.
It bridges runtime metrics, ethics, and automation: an observer that acts, and an actor that never stops observing.

“Observability with teeth” is not a slogan here; it’s a design discipline.
Ben’s tendrils reach every subsystem not to dominate, but to ensure each survives, adapts, and reports truthfully under stress.

That is the end of the writeup. I’m focusing on ben in order to court more collaborators.