DATA PROTECTION & SOVEREIGNTY

Governed Event Streams: Live and Replayable

A shared event bus asks you to trust the topic — if you can see it, you receive it. Pulse re-authorizes every row at the moment of delivery, on a stream your project owns outright.

September 2026 · v1.0 · 15 min read

01 The thesis: own the stream, trust every delivery
02 Stream: the event bus your project owns
03 Feed: seeing a topic is never enough to receive a row
04 The fan-out: one shared ring per topic, not one query per subscriber
05 Authorization that scales with the fan-out, not against it
06 Alerting: Feed's machine-facing sibling
07 The cryptographic foundation beneath the ownership claim

Executive summary

Pulse is Kilimanjaro's Data Intelligence plane, built from four products: Lakehouse and Stream for storage, Compute and Feed for activity. This paper covers the pair built for events in motion — Stream, the event bus a project owns outright, and Feed, the live plane that delivers those events, and everything else that happens inside a project, to a human dashboard or a downstream system in real time. Where the other two Pulse products are the subject of the sibling paper 'Analytics Without the Warehouse,' this one is about ownership and authorization at the speed of a live event.

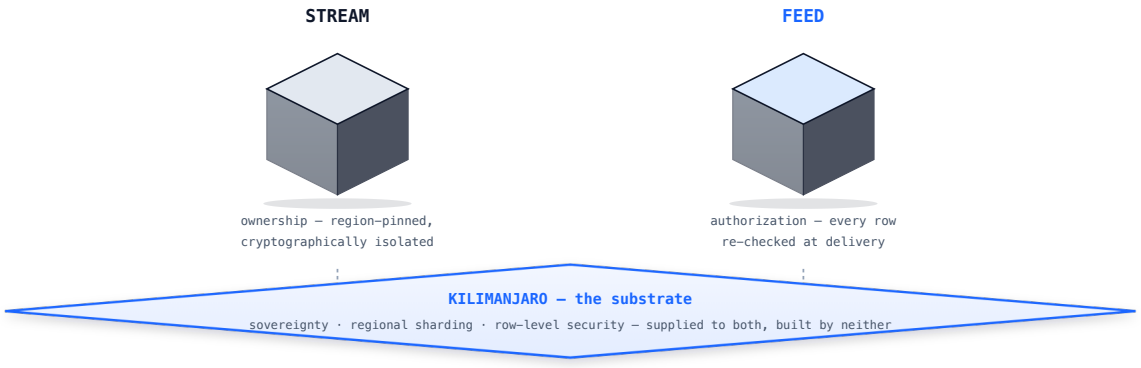
For ownership, the consequence is that a project's event stream is region-pinned to its own residency set and isolated from every other tenant cryptographically, on the bus itself, not by an access-control list bolted onto a shared account — and retention is something a project configures per stream, not a fixed default it inherits. For authorization, the consequence is that a topic binding is never treated as sufficient on its own: every row is re-checked against row-level security at the exact moment it's delivered, so a subscriber who can see a topic still only receives the rows they're allowed to see.

The paper works through both mechanisms in turn — the exact key hierarchy and account topology that make Stream's isolation cryptographic rather than logical, and the ring-buffer fan-out and compiled authorization predicate that let Feed serve a reconnect storm without turning it into a database storm — before closing with the same candor this series leads with everywhere else: what a claim covers, and where its edge is.

01 The thesis: own the stream, trust every delivery

A shared event bus is a rental agreement dressed up as infrastructure: you get a topic, you get an access-control list, and every guarantee past that point is the vendor's operational promise, not a property of the system. Kilimanjaro's Data Intelligence plane, Pulse, treats event streaming as two separable products with two separable guarantees.

Stream is the event bus itself — a customer-owned, region-pinned, cryptographically isolated JetStream instance that a project provisions, publishes to, and replays from directly. Feed is the live delivery plane on top of it — the surface a human dashboard or a downstream system actually subscribes to, where every row is re-authorized at the exact moment it's about to leave the building, not once when the subscription was created.



Stream and Feed sit on one inherited substrate rather than each re-implementing sovereignty for itself. Stream is the ownership claim — a region-pinned, cryptographically isolated bus. Feed is the authorization claim — every row re-checked at the moment of delivery. Kilimanjaro supplies the sovereignty, regional sharding, and row-level security that make both claims true.

The two claims compose but don't depend on each other. Stream's claim is about custody: a project doesn't rent a topic on someone else's shared cluster, it owns a stream that is region-pinned to its own residency set and isolated from every other tenant by cryptography, not by an access-control list layered on top of a shared account.

Feed's claim is about authorization: a Kafka-style topic ACL answers exactly one question — can this subscriber see this topic — and Pulse treats that question as necessary but not sufficient. Every row that crosses the wire is re-checked against row-level security at the moment of delivery, so a subscriber who can see a topic still only receives the rows it's authorized to see, row by row, live.

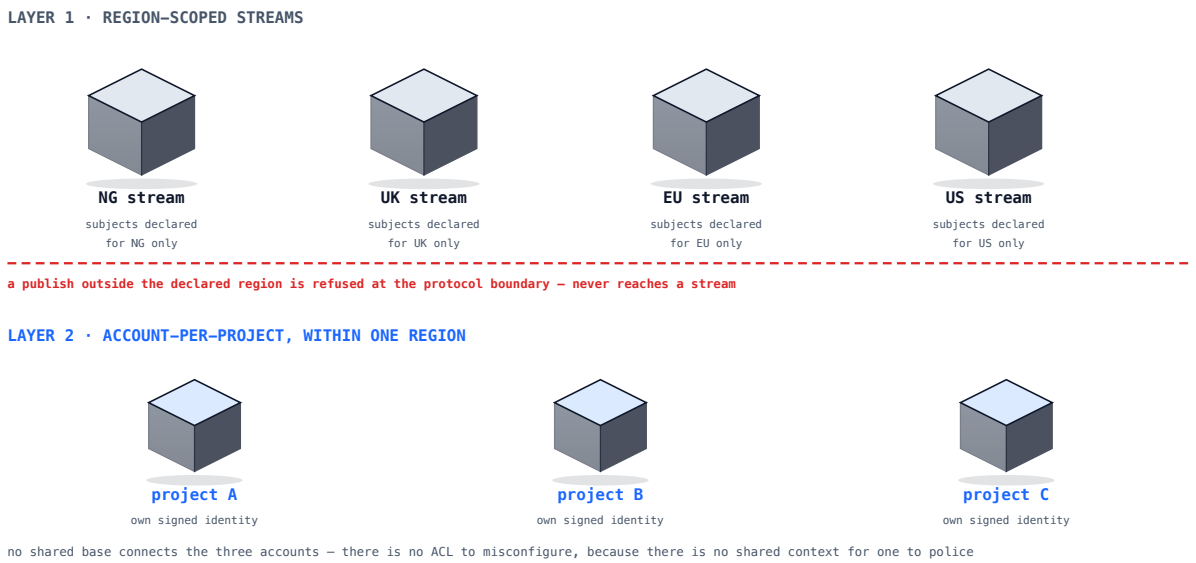
Both products inherit Kilimanjaro's sovereignty from the substrate rather than re-implementing it — regional sharding, per-tenant cryptographic isolation, and row-level security are properties of the platform beneath Stream and Feed, not features either product built for itself. That governance is a property of the substrate itself, applied identically to every project on it — never a capability a project has to configure its way into, and never a guarantee that varies by how a project is provisioned.

02 Stream: the event bus your project owns

Stream is built on NATS JetStream, not a bespoke bus — a single, self-hostable binary with a standard client API, so a project isn't locked into a proprietary wire protocol to get a sovereign event bus. A strict internal port around the bus client keeps the specific engine swappable — a future move to a different backing store would be a driver change, not a rewrite of every product built on top of it.

Against a hosted-only bus like a managed Kafka cluster or a service like Pusher or Ably, this is the lead differentiator: the bus itself is something a customer could stand up and inspect, not a black box a vendor operates on their behalf.

Isolation is architected at two layers, and both are cryptographic rather than merely logical. First, region: every serving region runs its own JetStream stream, and every subject a project publishes to is namespaced under that project's home region. A publish call for a region outside the project's declared residency set is rejected at the protocol boundary — before it ever reaches a stream, not caught by an audit sweep afterward. A subject's events physically cannot exist outside the region their project has declared for them.



Isolation is enforced at two independent layers, both cryptographic rather than logical: every region holds its own stream, with a project confined to it by residency; and every project holds its own JetStream account, a distinct signed identity with no shared context another project could see through.

Second, tenancy: Stream provisions one JetStream account per project, and a JetStream account is itself a distinct, signed cryptographic identity — not a row in a permissions table. Two projects sharing the same physical cluster have no shared account context to see through; there is no ACL to misconfigure, because there is no shared namespace for an ACL to police in the first place. The key hierarchy that mints and protects that account identity is set out in 'The cryptographic foundation beneath the ownership claim.'

Retention is a first-class, per-stream configuration, not an afterthought bolted onto a fixed default. A project declares how long each stream retains its events along a spectrum from a short, live window measured in minutes to same-day replay measured in hours to open-ended retention measured in days — the choice belongs to the stream's actual purpose, a live telemetry feed and a compliance-replay log have no reason to share a retention policy.

Because account-per-project isolation is native to JetStream, running many streams under one project is the natural case, not a special one, and blast radius is bounded at two independent layers rather than one: the project's account identity itself carries a fixed ceiling, set once at account creation and never reissued; and a second, independently-evaluated bound governs how much of that ceiling is actually in active use at any moment, checked on every request rather than baked into the identity. Separating the two means a project's cryptographic identity on the bus is permanent and never has to be re-minted just because its operational footprint changes — an identity is a fact about who a project is; a bound is a fact about what it's doing right now, and the architecture keeps the two from being the same number.

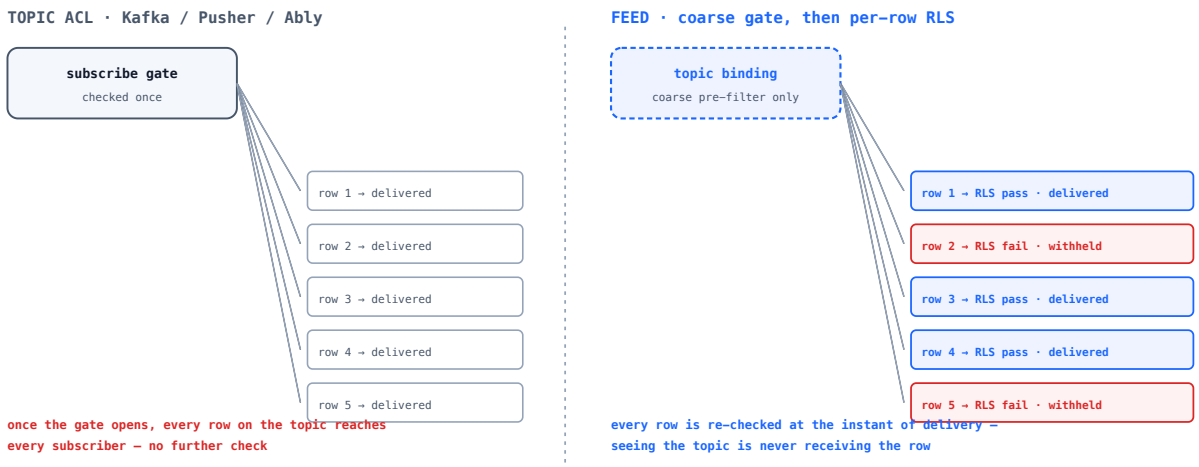
Layer	What it bounds	When it changes
Identity ceiling (account credential)	A fixed maximum baked into the project's bus identity at creation	Not re-minted in response to usage/quota changes
Operational bound (gateway-enforced)	Active usage against that ceiling, evaluated per request	Continuously, as a project's actual footprint changes

Two independent limit layers — a fixed identity ceiling set once at account creation, and a separately-evaluated operational bound checked on every request.

03 Feed: seeing a topic is never enough to receive a row

The property that makes Feed different from a topic-based pub/sub system isn't speed — it's where authorization happens, and how often it's re-evaluated. A Kafka, Pusher, or Ably topic ACL answers one question up front, at subscribe time: is this client allowed to see this topic. Once the answer is yes, every message published to that topic reaches every subscriber who passed the check — the topic boundary is the authorization boundary, and it's evaluated exactly once, at connection time.

Feed treats a stream or topic binding as a coarse pre-filter only, never as the authorization decision itself. The actual decision — the project's row-level security policy — is re-evaluated against every single row, at the exact moment that row is about to leave the server for that specific subscriber's connection.



A topic ACL is a single gate, checked once at subscribe time — once open, every message on the topic passes through it. Feed treats that same gate as a coarse pre-filter only: every row behind it is re-checked against row-level security at the instant of delivery, so seeing the topic and receiving the row are never the same decision.

This is a design invariant, holding for every event that flows through Feed, for every topic.

In practice it means a dashboard, an autonomous agent, or a downstream service can subscribe broadly to a topic representing everything happening in a project and still only ever receive the rows its own row-level security permits — the identical guarantee a direct, authenticated database query gets, carried through to a push-based, server-initiat-

ed delivery surface that most systems treat as an authorization-free zone the moment a subscription is granted. A subscriber whose access is revoked mid-stream stops receiving rows on the very next event, not on its next reconnect.

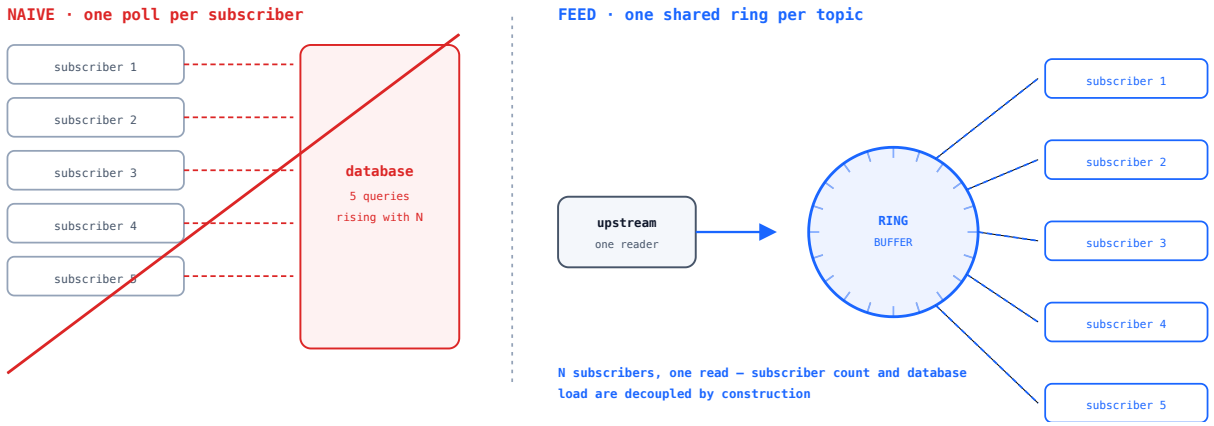
Delivery itself is deliberately one-way: Feed is built on Server-Sent Events, not a bidirectional WebSocket. Telemetry flows server to client; nothing flows the other way over this surface. That is a correctness and security choice, not a limitation — SSE is HTTP-native, so a dropped connection auto-reconnects, resumes from a last-event marker the client already holds, and the whole surface composes for free with the same authentication, network-allowlisting, row-level-security, and edge stack every other Kilimanjaro API already runs behind, because it is that stack, not a parallel one.

An agent that needs to converse with the platform bidirectionally does that over Compute's own Streamable HTTP surface — a different product, a different contract — never by pushing messages back over Feed.

04 The fan-out: one shared ring per topic, not one query per subscriber

A naive real-time layer polls the database on behalf of every open connection — ten subscribers means ten queries on a timer, and a fleet restart that reconnects ten thousand clients at once turns into ten thousand queries landing on the database in the same second, at the exact moment the database is least able to absorb them. Feed's fan-out removes subscriber count from that equation entirely. Per topic, per replica, there is exactly one live upstream subscription — a single reader that consumes new events once and writes them into an in-process ring buffer sized to hold a bounded, recent window of that topic's history.

Every subscriber to that topic, whether there are ten or ten thousand of them, reads from that same ring. Adding the ten-thousandth subscriber costs one more reader against memory already resident in the process; it does not cost the database a tenth query, because the database was never in the request path for delivery to begin with — publish path and read-fan-out path are structurally different circuits that happen to share a topic name.



A naive real-time layer polls the database once per open connection, so subscriber count and database load rise together — a ten-thousand-client reconnect becomes ten thousand simultaneous queries. Feed's fan-out reads each topic once, through a single shared upstream per topic per replica, into an in-process ring buffer every subscriber to that topic tails — so subscriber count and database load are decoupled by construction.

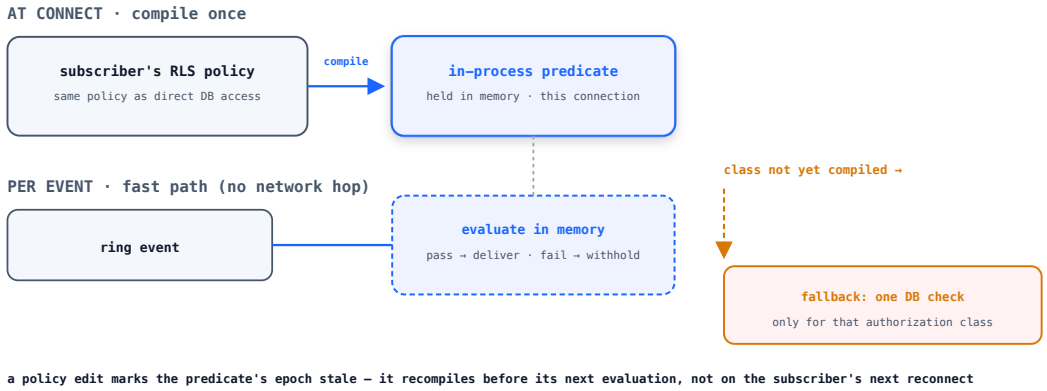
Reconnection gets a defined answer instead of a shrug, because the ring turns 'did I miss anything' into an arithmetic question rather than an open one. A client that drops and reconnects within the ring's retained window resumes from its own last position inside that ring — a cursor comparison, not a re-scan.

A client that has been gone longer than the ring retains gets one of two defined outcomes depending on the topic's nature, never a silent gap it has no way to detect: for audit-family topics, the platform's own signed, hash-chained audit ledger — durable across any restart, unlike the bounded ring — replays the gap directly; for every other topic, the client receives an explicit signal to resync from source, which for most consumers means one authenticated read against the current state rather than trying to reconstruct a delta it was never guaranteed to see. A resync signal is a slower path, not a wrong one.

05 Authorization that scales with the fan-out, not against it

Removing the database from the fan-out's read path only solves half the scaling problem, because authorization still has to happen somewhere, for every subscriber, on every event. A naive implementation re-introduces exactly the cost the ring buffer was built to avoid: one row-level-security check, one database round-trip, per subscriber, per event — the same multiplication the fan-out just eliminated on the read side, now reappearing on the authorization side.

Feed's answer is to move the check out of the hot path rather than accept the multiplication. At connection time, a subscriber's active row-level-security policy — the same policy that governs its direct database access — is compiled once into an in-process predicate: a piece of evaluable logic held in memory for the life of that connection, not a query re-issued per event.



A per-event database check for row-level security would reintroduce the exact cost the ring buffer exists to remove. Feed instead compiles each subscriber's active policy into an in-process predicate once, at connection time, and evaluates every event against that predicate in memory; only the authorization classes the compiled predicate doesn't yet model fall through to a database check, and a policy change invalidates the cached predicate immediately.

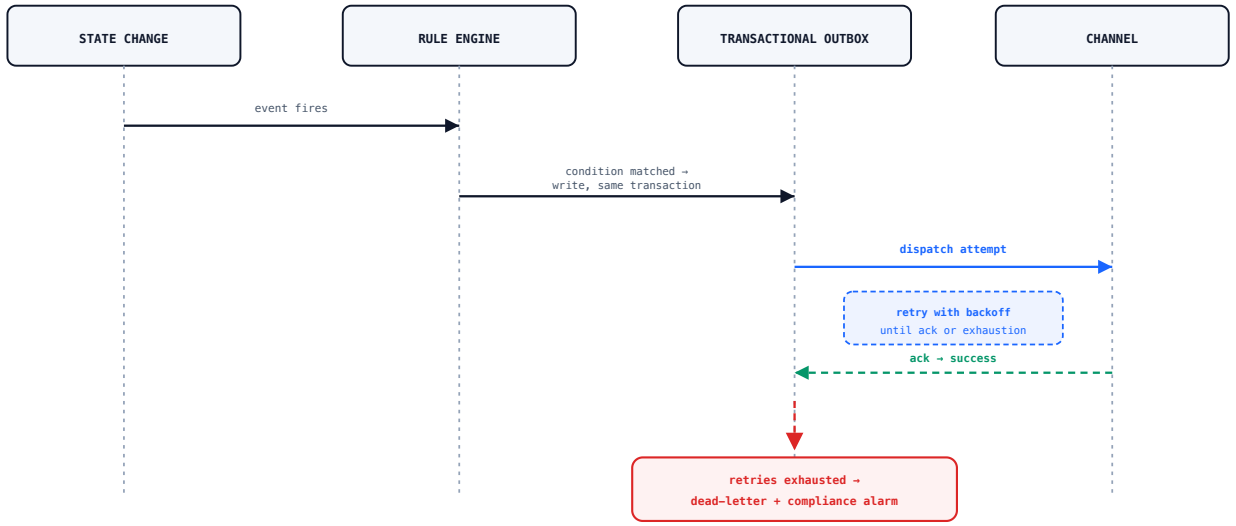
Every event that reaches the ring is then evaluated against every connected subscriber's compiled predicate in memory, in the same process that's already holding the ring — no network hop, no connection-pool contention, no query planner in the loop. The predicate compiler doesn't yet model every authorization class a project might use; where it can't compile a class in-process, that specific check — and only that check — falls through to a direct database read, so correctness never depends on compiler coverage.

And because a subscriber's access can change mid-connection, an epoch marker travels with every compiled predicate: the instant the underlying policy is edited, the predicate's epoch goes stale and it is recompiled before its next evaluation, so a revoked grant takes effect on the very next event, not on the subscriber's next reconnect.

06 Alerting: Feed's machine-facing sibling

Not every consumer of a live event is a human watching a dashboard. Alerting is the durable, machine-facing counterpart to Feed's human-facing stream: a rule engine that matches events — or their absence, since a rule can fire on 'no heartbeat in ten minutes' just as easily as on an event that did happen — applies routing, throttling, and de-duplication, and dispatches through a transactional outbox rather than a fire-and-forget call. Where Feed optimizes for a human watching a live surface, Alerting optimizes for the case where nobody is watching until the rule fires.

Delivery is at-least-once by construction, not by best effort: a dispatch is written to a transactional outbox in the same transaction as the state change that triggered it, so a crash between the two is structurally impossible — either both happened or neither did. From the outbox, retry with backoff carries a delivery until it succeeds or exhausts into a dead-letter path, and dead-lettering itself raises a compliance alarm rather than closing silently; a dropped alert is a visible failure, never an invisible one.



The dispatch path as a sequence: the triggering state change and the outbox write happen in one transaction, so a crash between them is structurally impossible. Dispatch retries with backoff until the channel acknowledges; only after every retry is exhausted does a delivery dead-letter, and dead-lettering itself raises a compliance alarm rather than closing silently.

Two channels are native to the outbox: email through a tenant's own bring-your-own SMTP configuration, templated, rate-limited, and throttled so one incident can't fan out into a thousand messages; and signed HTTPS webhooks, guarded against SSRF and DNS-rebinding and restricted to a per-tenant allowlist.

Isolation follows the same per-project account boundary Stream uses: each project's alerting consumer reads only its own durable subscription and holds only its own windowed de-duplication state, physically fenced from every other project's, so one tenant's alert volume or backlog can never starve, delay, or leak into another's.

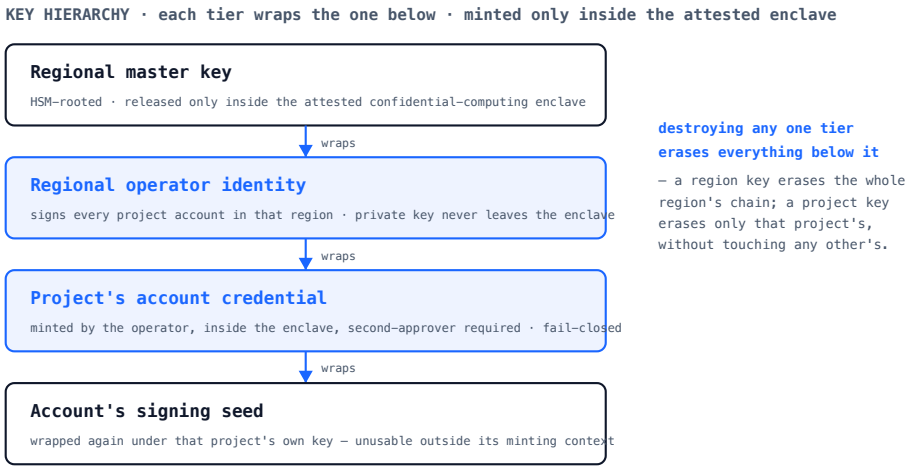
Capability	What it is
Rule matching	Filter + condition (event, threshold, or absence) + routing + throttle/dedup, versioned per org/project
Delivery guarantee	At-least-once via a transactional outbox: retry with backoff, dead-letter on exhaustion, a compliance alarm the instant that happens
Channels	Bring-your-own-SMTP email, templated and rate-limited; signed HTTPS webhooks, SSRF-guarded and allowlisted
Isolation	Per-project consumer and windowed dedup state, fenced by the same account boundary as the project's Stream

What Alerting does, end to end.

07 The cryptographic foundation beneath the ownership claim

Everything this paper calls owned and isolated rests on a specific key hierarchy. The hierarchy has four tiers, each wrapping the one beneath it. At the floor tier, the regional master key is a software-protected sovereign vault secret, wrapped in-process, with no HSM claim and no attestation claim.

At the Secure Key Release (SKR) custody tier, that root is instead hardware-rooted — a regional master key in an HSM — and never leaves it in the clear; it is releasable only inside an attested confidential-computing enclave, the same trust boundary the rest of the platform's key custody uses at that tier. Either way, that master key wraps a regional operator identity — the signing authority for every project's bus account in that region, minted once under a maker-checker, two-approver genesis ceremony inside the platform's controlled environment.



Each tier wraps the one below it. At the Secure Key Release custody tier, a hardware-rooted regional master key, releasable only inside an attested confidential-computing enclave, wraps the regional operator identity that signs every project's bus account; at the floor tier that root is a software-protected sovereign vault instead. Either way, the account's own signing seed is wrapped again under that project's key, and no tier can be minted or unwrapped outside its own trust boundary.

A project's account is then minted downstream of that operator identity: unwrapping the operator key and signing the project's account credential in one operation, chained to and signed by the already-attested operator root, and failing closed — an unattested request or an unavailable root produces no account, never a weaker or unsigned one as a fallback. This per-project mint does not itself require a separate second-approver ceremony; the two-party approval sits at the operator-root genesis, once per region. The account's own signing seed is then wrapped a fourth time, under a key scoped to that specific project, so the account credential itself is unusable outside the context that minted it.

Erasure inherits the same structure it's built on, because destroying a key and destroying access are the same operation at every tier. Destroying a region's operator key erases that region's entire bus trust chain in one act — no project in that region can be re-authenticated against it afterward. Destroying a single project's key renders that one project's account seed unrecoverable without touching any other project's chain, because the wrapping at that tier was never shared.

And a subject-level erasure reaches every tier a subject's data actually lives on: the live stream and its own cold, compacted retention tier are purged together, by the same key-destruction primitive that erases the rest of that subject's footprint across the platform — one action, not two deletion jobs in two systems that have to agree with each other to be correct.

What we disclose

Feed's in-process authorization predicate models a subset of the row-level-security classes a project can define; the classes it doesn't model resolve through a direct database check instead of the in-process fast path, correctly but at the database's speed. The architecture decouples subscriber count from database load; the paper states no subscriber ceiling.

There is no subscribable event bus for the platform's own internal vault operations, and this paper does not claim one. Vault-related activity is recorded on the signed audit ledger and read back through an authorized, row-level-secured query — a pull, not a push — and that remains the design until a genuine internal need for a subscribable vault-operations bus exists.

Stream is transport and a retention buffer, not the system of record: the platform's signed, hash-chained audit ledger remains the authoritative history of what happened in a project, and nothing in this paper should be read as displacing it. 'Unlimited' is not a word this paper uses for stream storage — a project's identity ceiling is fixed and set once, but it is a real, server-set number, not an infinite resource.

See how ownership and re-authorization compose for your data

Talk to a Sovereign Engineer about which of your real-time and event-driven workloads need a stream you own outright, a feed that re-checks every row at delivery, or both.

Talk to a Sovereign Engineer →

See how Pulse works