The Nervous System

How events flow, how handlers execute, and why the messaging layer matters

February 2026

The event loop

A vault is a program that receives events, acts on them, and sends events back. That sentence appeared in an earlier document and it was deliberately simple. Now it is time to look inside.

The vault manager sits inside a hardware-isolated enclave. It has access to the vault's database, the user's keys, and a connection to the messaging layer. It subscribes to specific topics on the messaging layer and waits. When an event arrives on one of those topics, the vault manager reads it, determines what kind of event it is, checks it against the relevant contract and policies, and if everything passes, invokes the appropriate handler. The handler does its work — signing, encrypting, decrypting, looking up credentials, updating state — and the vault manager sends the result back out on the messaging layer.

That is the entire runtime. Events arrive. The vault manager validates and routes. Handlers execute. Results depart. There is no REST API, no HTTP server, no open ports. The vault's only interface with the outside world is the messaging layer. Everything enters and exits through events.

> *The vault has one interface: the messaging layer. Events come in on topics the vault subscribes to. Events go out on topics the vault publishes to. There are no open ports. There are no API endpoints. There is only the event loop.*

The messaging layer

The messaging layer is the communication fabric that connects everything on the vine: your mobile app to your vault, your vault to other vaults, services to your vault, and system infrastructure to all of the above. In

VettID’s implementation, this layer is built on NATS — a lightweight, high-performance messaging system designed for exactly this kind of distributed communication.

But before we talk about NATS specifically, it helps to understand what the messaging layer needs to do. Any messaging system that serves as the vine’s nervous system must support a few fundamental properties.

What the messaging layer must do

Subject-based routing: Messages are addressed to topics, not to IP addresses or endpoints. A topic like “user-12345.forVault.payment.request” tells the messaging layer exactly where the message should go. The vault subscribes to its topics and receives only the messages intended for it.

Publish and subscribe: The core pattern is pub/sub. A service publishes a payment request to the user’s topic. The user’s vault is subscribed to that topic and receives the message. The service does not need to know where the vault is, what infrastructure it runs on, or whether it is currently online. It publishes the event and the messaging layer handles delivery.

Delivery guarantees: Some events are fire-and-forget: a profile update notification, a status ping. Others are critical: a payment request, a document signing request, a contract update. The messaging layer must support persistent delivery for critical events so that if a vault is temporarily offline, the event is waiting when it comes back.

Scoped permissions: Not everyone can publish to every topic. A service can only publish to its designated topics in your message space. It cannot subscribe to your other topics. It cannot observe your messages with other connections. The messaging layer must enforce topic-level permissions so that the communication channels themselves are secure, independent of payload encryption.

Encrypted payloads: The messaging layer transports encrypted messages. Every sensitive payload is encrypted end-to-end before it is published. The messaging infrastructure sees envelopes, not contents. Even if the messaging layer itself were compromised, the payloads are opaque without the recipient’s keys.

Two communication spaces

The vine uses two distinct communication spaces, each with its own purpose and security boundary.

Space	Purpose	Who talks here
OwnerSpace	Secure communication between your mobile app and your vault. This is the private channel where you issue commands and receive responses.	Your app and your vault only. Nobody else can publish or subscribe to your OwnerSpace topics.
MessageSpace	Cross-vault messaging. This is where connections reach your vault: friend messages, service requests, contract updates, call signaling.	Your vault subscribes. Connections and services publish to designated topics. Services are restricted to their own topic path and cannot observe other traffic.

The separation is deliberate. Your app talks to your vault on OwnerSpace — a completely private channel. The outside world talks to your vault on MessageSpace — a channel where each connection is scoped to its own topic and cannot see anyone else's messages. The vault bridges the two: it receives commands from you on OwnerSpace, receives events from the world on MessageSpace, and routes everything through the event loop.

> *Two spaces. OwnerSpace is you talking to your vault — private, direct, nobody else listening. MessageSpace is the world reaching your vault — scoped, permissioned, each connection isolated to its own topic. The vault sits between them, processing everything through the event loop.*

How an event flows

Let us trace a concrete event from beginning to end: a retailer requests your shipping address.

The journey of a shipping address request

1. The service publishes: The retailer's service vault publishes a data request event to your MessageSpace topic. The event is encrypted end-to-end with your vault's public key. The messaging layer routes it to your vault’s subscription. The retailer does not know where your vault is running. It published to a topic. The messaging layer did the rest.

2. Your vault receives: The event arrives on your vault’s MessageSpace subscription. The vault manager decrypts the envelope, reads the event type (data.request), and identifies the originating connection (Acme Retail, connection ID abc-123).

3. Contract check: Before anything else happens, the vault manager checks the request against the connection’s contract. Is “shipping address” a declared field? What tier is it? Is the contract current? Are we within rate limits? If any check fails, the request is blocked and logged. The handler is never invoked.

4. Handler invocation: The contract says the shipping address is an on-demand field. The vault manager invokes the data access handler. The handler reads the shipping address from the personal namespace, packages it into an encrypted response, and hands it back to the vault manager.

5. Response published: The vault manager encrypts the response with the connection’s shared key and publishes it to the service’s topic. The retailer’s vault receives the encrypted response, decrypts it, and has the shipping address. The entire interaction is logged in the audit trail: what was requested, by whom, when, under what contract version, and what was returned.

That is the entire flow. No HTTP request hit a server. No REST API was called. No database query ran against a centralized user store. A service published an event. The messaging layer routed it. The vault validated, processed, and responded. All through events on topics.

What if it is a consent field?

If the requested field were at the consent tier instead of on-demand, the flow changes at step four. Instead of invoking the data access handler immediately, the vault manager creates a pending approval request and publishes a notification to your app on OwnerSpace. Your app shows you: Acme Retail is requesting your date of birth for age verification on alcohol purchases. You see the field, the purpose, and the requesting connection. You approve or deny. If you approve, the vault manager invokes the handler and the response flows back to the service. If you deny, the service gets a rejection. The event was waiting for your decision, and nothing happened until you decided.

> *Every event follows the same path: arrive on a topic, decrypt, validate against the contract, invoke the handler if approved, encrypt the response, publish. The path is the same whether the answer is automatic or requires your explicit consent. The contract determines which.*

What handlers actually are

A handler is a program that performs a specific operation inside the vault's enclave. Not a plugin. Not a script. A standalone executable built and deployed by the vault provider, living alongside the vault manager in the same hardware-isolated environment.

In VettID's implementation, handlers are separate Go executables that exist in the enclave independently of the vault manager. The vault manager invokes them when the right event arrives, but the handlers are not compiled into the vault manager binary. They are their own programs. This means any vault manager can call them, and the code architecture is modular — the dispatcher and the workers are separate concerns. However, handlers and the vault manager share the same enclave and the same attestation measurement. They are part of one attested image. Change a handler and the enclave's measurement changes, which means re-attestation. The independence is architectural, not operational: separate programs, same trust boundary.

Types of handlers

Each handler corresponds to a specific event type. When an event arrives, the vault manager matches the event type to the registered handler and invokes it.

Event type	Handler	What it does
payment.request	Payment handler	Reads the payment credential from the personal namespace, signs the transaction with the user's key, returns the signed authorization.
document.sign	Signing handler	Reads the document hash, signs it with the user's identity key inside the enclave, zeros the key from memory, returns the signature.
identity.verify	Verification handler	Generates a cryptographic proof that the user holds a specific credential without revealing the credential itself.
data.request	Data access handler	Reads the requested field from the appropriate namespace, packages and encrypts the response.
messaging.send	Messaging handler	Encrypts the message with the recipient's public key, publishes to their MessageSpace topic.
auth.challenge	Authentication handler	Responds to a service's authentication challenge by signing the nonce with the connection's key.

This table is not exhaustive. The set of handlers is what defines the vault's capabilities. A vault provider that builds a healthcare handler for consent management has different capabilities than one that builds a financial handler for multi-party transaction signing. The handlers are the product. The vault manager and messaging layer are the platform.

Handler manifests

Every vault publishes a list of its supported handlers in its profile. This is the handler manifest: a declaration of what event types the vault can process, what version of each handler it runs, and what permissions each handler requires.

When a connection looks at your profile, it sees your handler manifests alongside your credential metadata. The retailer sees that your vault supports payment.request and identity.verify. The employer sees that your vault supports document.sign and auth.challenge. The connection knows what your vault can do before it asks. If a service publishes an event type your vault does not support, the event is rejected — not silently dropped, but explicitly rejected and logged.

The manifest is also how capabilities evolve. When your vault provider ships a new handler — say, a location.share handler for real-time location tracking — it ships as part of a new enclave image with a new attestation measurement. Your vault's manifest updates and your connections can see the new capability. The vine is programmable because the handlers are deployable, and the manifests make the capabilities discoverable.

> *Handlers are standalone executables inside the enclave. They are the vault's capabilities. The vault provider builds them, deploys them, and declares them in the handler manifest. Any vault manager can invoke them. Connections see what your vault can do. The vine is programmable because the handlers are.*

Where vault providers differentiate

The vault manager, the database, and the messaging layer are the foundation. Every vault provider needs them, and they work the same way regardless of what the vault does. The handlers are where providers build their product.

This is by design. The foundation is the boring part — reliable, standardized, interchangeable. The handlers are the interesting part — specialized, differentiated, where competitive advantage lives.

VettID’s focus

VettID builds handlers for the critical universal operations that touch every user’s secrets, identity, and safety: authentication, authorization, transaction signing, secure messaging across text, voice, and video, and location tracking. These are the capabilities that matter most for the individual — the operations where cryptographic enforcement and hardware isolation are non-negotiable.

Where others could specialize

Healthcare: Handlers for patient consent management, medical record sharing authorization, insurance verification, clinical trial enrollment. The consent handler alone is a product: managing who can see which records, under what conditions, with what expiration, all enforced in the enclave.

Financial services: Handlers for multi-party transaction authorization, regulatory reporting, cross-border compliance, institutional key management. A multi-signature handler that requires three-of-five vault approvals before releasing funds is a differentiated capability.

Enterprise: Handlers for organizational policy enforcement, role-based access delegation, compliance audit integration, fleet device management. An enterprise vault provider’s policy handler could enforce separation-of-duties rules that prevent any single employee from approving their own expense reports.

The vault manager routes the events. The database stores the state. The messaging layer delivers the messages. But the handler decides what happens when a payment request arrives, when a document needs signing, when a consent form needs approval. That decision — how to handle the event — is where the value lives and where providers compete.

> *The foundation is standard: vault manager, database, messaging layer. The handlers are the product. A healthcare provider builds consent handlers. A financial provider builds multi-signature handlers. An enterprise provider builds policy handlers. Same foundation, different capabilities, and the user picks the vault that matches their life.*

The interoperability question

Everything in this document works if there is one vault provider. But ZKT is a pattern, not a product. The goal is multiple vault providers, each specializing in different capabilities, all connected on the same vine. And that raises the most important question in the architecture: how do vaults from different providers talk to each other?

The problem

If every vault provider builds its own messaging layer, the vine fragments. My VettID vault cannot talk to your healthcare vault if we use different wire formats, different topic conventions, different encryption envelopes, different delivery semantics. We end up with walled gardens — each provider’s users can talk to each other, but not across providers. That is not a vine. That is a collection of isolated networks pretending to be connected.

The handlers can be different. That is the whole point of provider differentiation. But the layer underneath them — the messaging layer, the event format, the routing conventions, the envelope structure — must be common. This is the common language that vaults speak to each other. Without it, there is no interoperability. Without interoperability, there is no vine.

What needs to be standardized

Not the handlers. Not the database. Not the enclave implementation. These are where providers differentiate and compete. What needs to be standardized is the communication layer: the narrow interface between vaults that makes cross-provider messaging possible.

Component	What the standard must define
Event format	The structure of an event: event ID, event type, timestamp, encrypted payload. Every vault must produce and consume events in the same format.
Envelope encryption	The encryption scheme for payloads in transit. Algorithm, key exchange method, nonce handling. Every vault must encrypt and decrypt envelopes the same way.
Topic conventions	The naming structure for topics: how to address a specific vault, how to scope service topics, how to separate owner communication from cross-vault messaging.
Routing rules	How events are addressed and delivered: subject-based routing, permission scoping, which topics a connection can publish to and subscribe on.
Delivery guarantees	Which events require persistent delivery (critical operations like signing and payment) and which are best-effort (notifications, status updates).
Contract format	The structure of connection contracts: field specifications, tier definitions, retention policies, permissions, rate limits. Every vault must be able to read, present, and enforce contracts from any provider.
Connection handshake	The protocol for establishing a new connection between two vaults: profile exchange, key negotiation, credential scoping, consent flow, and contract presentation. If two vaults from different providers cannot agree on how to connect in the first place, nothing else in the standard matters.

Everything above the standard is provider territory. The handlers, the database engine, the enclave implementation, the mobile app experience, the user interface for managing connections — all of it. The standard defines the wire. The providers build the product. And the connection handshake is the front door: if two vaults cannot agree on how to establish a connection, nothing behind it matters. This is the same kind of problem that LEASH addresses for agent credential handling — a vendor-neutral protocol that lets any two compliant implementations work together without prior coordination.

Why NATS fits

VettID uses NATS with JetStream as its messaging layer. This is one implementation, not a prescription. But it is worth understanding why it fits the requirements so well, because any standard will need to support the same properties.

NATS is lightweight and built for pub/sub communication with subject-based routing. Topics map naturally to vault namespaces: a subject like "MessageSpace.user-12345.fromService.acme-retail.data.request" tells you exactly who is being addressed, who is sending, and what type of event it is. The subject hierarchy is the routing table.

JetStream adds persistence. Critical events — payment requests, signing requests, contract updates — are stored in streams and delivered when the vault comes online, even if it was offline when the event was published. Best-effort events use core NATS without persistence. The same infrastructure handles both, with the delivery guarantee declared per event type.

NATS supports scoped credentials and topic-level permissions natively. A service connection gets a credential that can only publish to its designated topics in your message space. It cannot subscribe to your other traffic. It cannot observe your owner space. The permission enforcement happens at the messaging layer, before the event reaches your vault. This is defense in depth: even if a service's credential were compromised, the scope of damage is limited to the topics that credential can access.

The argument is not "everyone should use NATS." The argument is: here is what the messaging standard needs to support — subject-based routing, pub/sub, persistent delivery for critical events, scoped permissions, encrypted payloads. NATS with JetStream satisfies all of these. A standard based on these requirements would allow NATS-based providers to interoperate with providers using other messaging systems that satisfy the same properties.

The path to a standard

VettID is pursuing the same approach here that it takes with LEASH for agent credential handling: propose a vendor-neutral specification with VettID as an example implementation. The specification defines the connection handshake, the event format, the envelope encryption, the topic conventions, the routing rules, the delivery guarantees, and the contract format. VettID's NATS implementation is one system that satisfies the specification. Other providers can satisfy it with different infrastructure.

This is not altruism. It is strategy. A standard that enables interoperability makes every vault provider more valuable, including VettID. A user who chooses VettID for its authentication and messaging handlers can still connect to a healthcare provider's vault for medical consent management, or an enterprise provider's vault for organizational access. The vine grows across providers, not within them. And the larger the vine, the more valuable every node on it becomes.

The alternative is fragmentation: each provider builds its own island, users are locked to one ecosystem, and the promise of ZKT — that you choose the vault that matches your life — is broken at the infrastructure level. A messaging standard prevents that. It keeps the vine open while letting providers compete on what matters: the handlers, the experience, and the capabilities.

> *The handlers are the product. The messaging layer is the commons. Standardize the wire, and every vault provider can talk to every other. Fragment the wire, and ZKT becomes a collection of walled gardens. The vine needs a common language. The standard defines it. The providers speak it. The users benefit.*