Event Store Selection for the Janus Backend

Purpose of an Event Store

The Janus architecture uses a slot-based model where each component fills a well-defined role in the stack. A **document store** captures unstructured data and a **relational database** holds normalized state. An **event store** provides a *chronological record* of every significant change in the system. In an event-sourced design, events are the source of truth and allow the system to reconstruct state by replaying events. Event sourcing improves auditability, avoids lost history and simplifies distributed workflows【393156474872709†L42-L56】. Each event is appended to an *append-only log* and is never mutated; consumers build projections or materialized views from the event stream【393156474872709†L84-L104】. The Janus platform needs an event store to:

* Record all agent actions, system events and user commands to enable *audit trails* and reproducibility.
* Support *append-only writes* with immutable events to maintain the full history【450349005886131†L120-L124】.
* Provide *efficient retrieval* of events for a particular entity or stream (e.g., a user, project or agent run)【640399355648082†L148-L166】.
* Offer *concurrency control* so that multiple writers cannot overwrite the same stream state【640399355648082†L168-L171】.
* Run *self-hosted* in offline/edge environments, with high availability and security (TLS, authentication, encryption at rest).

Requirements and Evaluation Criteria

1. **Durability & Ordering** – Each event must be persisted in order and remain immutable. The store should support replaying events to reconstruct state.
2. **Write Throughput & Read Performance** – Even though Janus is designed for a single developer initially, the event store must scale to handle many events per second when multiple agents run concurrently. Efficient indexing per stream is important for fast reads【640399355648082†L148-L166】.
3. **Concurrency Control** – Optimistic concurrency (expected version) or transactional isolation is needed to avoid race conditions when writing events【640399355648082†L168-L171】.
4. **High Availability & Resilience** – Clustered deployment and replication are essential for fault tolerance【450349005886131†L137-L144】.
5. **Integration & Ecosystem** – The solution should integrate well with Python and existing stack components (PostgreSQL, RabbitMQ, Redis), and ideally run in containers orchestrated by Kubernetes.
6. **Security & Access Control** – TLS/SSL encryption, authentication, authorization and encryption at rest. A smaller attack surface and support for offline environments are desirable.
7. **Operational Complexity** – Ease of setup and management, especially for a solo developer. Heavy operational burden is undesirable.

Candidate Technologies

EventStoreDB

Description: EventStoreDB is a specialized database built specifically for event sourcing. It stores data as *streams of immutable events* and indexes them by stream ID. Developers can reconstruct an entity’s state by replaying its events【450349005886131†L103-L124】. The database supports real-time streaming and clusters for high availability【450349005886131†L129-L144】.

Key capabilities

| Capability | Notes |

|---|---|
Immutable events & append-only log	Once appended, events cannot be modified. This immutability guarantees data integrity and provides a reliable audit trail【450349005886131†L103-L124】.
Per-stream indexing & high write throughput	EventStoreDB can handle ~15 000 writes/sec and efficiently fetch events for a stream thanks to indexes【640399355648082†L148-L166】.
Optimistic concurrency control	Writers specify the expected version when appending events; mismatched versions cause the write to fail, preventing lost updates【640399355648082†L168-L171】.
Real-time subscriptions & streaming	Clients can use live subscriptions, catch-up subscriptions or persistent subscriptions to receive events in real time【640399355648082†L173-L176】.
High availability & clustering	Supports clustered deployments with replication for fault tolerance【450349005886131†L137-L144】.
Security	Provides built-in TLS encryption and authentication (user/password or certificates) and supports fine-grained access control.

Pros

* Purpose-built for event sourcing with immutability and per-stream indexes.
* Scales to high write throughput and supports clustering for high availability.
* Native features like subscriptions and optimistic concurrency simplify application logic.
* Integration libraries exist for many languages including Python.

Cons

* A standalone database to deploy and manage; adds operational complexity compared to reusing existing Postgres.
* Though open source, advanced features are available under commercial license; community support is smaller than mainstream databases.
* Resource hungry relative to small single-node environments.

PostgreSQL as an Event Store

Description: Rather than adopting a specialized database, an event store can be implemented in an existing relational database. Each event is stored in a table with columns for `id`, `stream_id`, `version` and `data` (JSONB)【311468078120902†L462-L485】. Retrieving events for a stream uses a simple SQL query ordered by version【311468078120902†L495-L501】, and inserting a new event increments the version atomically within a transaction【311468078120902†L508-L520】.

Key capabilities

Capability	Notes
ACID transactions & durability	PostgreSQL provides ACID compliance【450349005886131†L178-L184】. Events can be inserted within a serializable or repeatable-read transaction to ensure correctness【311468078120902†L518-L539】.
Simple schema	A single table with `stream_id`, `version` and JSONB payload captures the event stream【311468078120902†L462-L485】. Additional indexes can be added on `stream_id` and `version` for efficient lookups.
Flexible JSON storage	Event data can be stored as JSONB, allowing events to contain arbitrary fields【311468078120902†L462-L485】.
Reuses existing infrastructure	No new database to deploy; uses the existing PostgreSQL instance already locked in the Janus stack.

Pros

* Reuses the locked-in PostgreSQL, minimizing operational overhead and cost.
* ACID transactions guarantee consistency and durability; concurrent writes can be serialized.
* Simplifies backup and security: Postgres supports TLS, row-level permissions and encryption at rest via pgcrypto or full-disk encryption.
* Good fit for moderate throughput systems where the volume of events is not extremely high; all state is local and offline-friendly.

Cons

* Performance may become a bottleneck at very high write rates; indexes on a large events table could impact query times.
* Application code must handle optimistic concurrency by reading the current max version and inserting the next version within a transaction 【311468078120902†L508-L520】.
* Lacks built-in subscription mechanism; event distribution must be implemented using the message broker (RabbitMQ) or Postgres LISTEN/NOTIFY.

Apache Kafka

Description: Kafka is a distributed messaging platform that stores data in append-only logs (topics) and is optimized for throughput. It is often used for event streaming and integrates with frameworks like Kafka Streams.

Key capabilities

Capability	Notes
High throughput & scalability	Kafka can ingest hundreds of thousands of messages per second and supports horizontal scaling through partitions 【640399355648082†L184-L200】.
Durable storage & replication	Topics are replicated across brokers to ensure durability and availability.
Rich ecosystem	Mature tooling (Kafka Streams, Schema Registry, ksqlDB) and broad community support.

Limitations for event sourcing

* **Limited per-stream indexing** – Kafka partitions messages by key; retrieving all events for a specific stream requires scanning a partition, which becomes slow as the number of events grows【640399355648082†L190-L204】.
* **No optimistic concurrency control** – Kafka cannot reject writes based on version; concurrency must be handled externally【640399355648082†L218-L223】.
* **Operational complexity** – Requires running a Kafka cluster (and often ZooKeeper) and managing partitions, retention policies and consumer groups.
* **Resource heavy** – Suitable for high-scale streaming systems but overkill for moderate workloads.

NATS JetStream

Description: JetStream extends the NATS messaging system with durable storage, message replay and streaming semantics. It offers low-latency pub/sub with persistent streams and is designed for microservices and real-time analytics【129778631843866†L33-L38】.

Key capabilities

Capability	Notes
Durable message storage & replay	JetStream stores messages in file- or

memory-based streams and allows consumers to replay events from any point in time【129778631843866†L46-L55】. |
Delivery semantics	Supports at-least-once and exactly-once delivery, durable subscriptions and ordered messages【129778631843866†L49-L55】.
High performance & low latency	JetStream is optimized for low latency (<20 ms) and high throughput for both persistent and non-persistent messages【129778631843866†L42-L45】.
High availability & scalability	Clustering and manual sharding enable horizontal scaling; streams can be sharded across nodes【129778631843866†L52-L107】.
Security	Provides TLS for secure communication and fine-grained access control【129778631843866†L57-L58】.
Simple API	Developer-friendly API and tooling; multi-tenancy support【129778631843866†L60-L64】.

Pros

* Excellent performance, low latency and good streaming semantics.
* Durable storage with replay and exactly-once delivery options.
* Strong security features and multi-tenant isolation.
* Lighter footprint than Kafka; can run on a small cluster.

Cons

* Operational complexity increases when managing clusters and manual sharding【129778631843866†L159-L165】.
* Smaller community and ecosystem compared with Kafka; fewer Python client libraries.
* Still primarily a messaging system; event sourcing requires conventions for stream naming and concurrency, and there is no built-in per-stream versioning.

Apache Pulsar

Description: Pulsar is a multi-tenant, high-performance distributed messaging and streaming platform. It supports both persistent and non-persistent topics and scales by partitioning topics across brokers【541665791063620†L70-L114】.

Key capabilities

Capability	Notes
Multi-tenant architecture	Pulsar supports tenants and namespaces with strong isolation【541665791063620†L70-L74】.
Persistent vs non-persistent topics	Messages can be durably stored on disk or kept in memory for transient communication【541665791063620†L96-L110】.
Flexible routing modes	Producers can route messages via round-robin, single partition or custom partitioning【541665791063620†L131-L146】.
Subscription modes	Consumers can use exclusive, failover or shared subscriptions for different delivery guarantees【541665791063620†L148-L187】.

Pros

* High performance and tiered storage; supports very large backlogs.
* Built-in multi-tenancy and flexible subscription models.

Cons

* Operationally complex: requires running brokers and Apache BookKeeper; heavier

than Postgres or JetStream.
* For event sourcing, retrieving all events for a specific stream still requires scanning partitions; concurrency control must be implemented externally.

Comparison and Recommendations

Solution	Durability & Concurrency	Performance	Security & Ops	Suitability for Janus
EventStoreDB	Immutable events, per-stream indexing and optimistic concurrency【640399355648082†L148-L170】	High write throughput (~15 k writes/s) and real-time streaming【640399355648082†L148-L176】	Built-in TLS, authentication and clustering; separate service to operate	Purpose-built for event sourcing; ideal for high-volume or multi-node deployments but adds a new database to manage
PostgreSQL event table	ACID transactions ensure consistency; version implemented via transactions【311468078120902†L508-L520】	Adequate for moderate workloads; indexes on `stream_id` support fast reads	Mature security features (TLS, row-level privileges); reuses existing infrastructure	Simple and low-overhead; best fit if event volume is moderate and we prefer to avoid operating additional services
Kafka	Durable log; partitions limit per-stream retrieval; no built-in versioning【640399355648082†L190-L204】【640399355648082†L218-L223】	Extremely high throughput; built for large clusters	Supports TLS and ACLs; heavy operations requiring ZooKeeper (or KRaft)	Overkill for Janus; poor per-stream access and concurrency control
NATS JetStream	Durable streams with replay; lacks explicit versioning	Low latency (<20 ms) and high throughput【129778631843866†L42-L45】	TLS and fine-grained permissions; manual sharding adds complexity【129778631843866†L159-L165】	Good for real-time streaming and microservices; extra service to manage and smaller ecosystem
Apache Pulsar	Durable topics; partitions used for streams; no per-stream versioning	High throughput; tiered storage; multi-tenant【541665791063620†L70-L114】	TLS and multi-tenant isolation; complex to run	Suitable for large multi-tenant streaming platforms, not necessary for Janus

Recommendation

Given Janus' **offline-first**, self-hosted ethos and moderate scale, **implementing the event store in the existing PostgreSQL database** is the most pragmatic choice. Postgres already serves as our relational store, and adding an `events` table with `stream_id`, `version` and `data` fields provides:

* **Low operational overhead** – no additional database to run; uses the same backup/restore and security procedures.
* **ACID durability** – ensures that events are persisted reliably and that concurrent writers cannot corrupt streams【311468078120902†L508-L520】.
* **Security** – Postgres offers mature TLS support, role-based access and full-disk encryption, reducing the attack surface.
* **Sufficient performance** – for a solo developer or small cluster, Postgres can handle thousands of events per second with proper indexing. If event volume becomes a bottleneck, the architecture can evolve toward EventStoreDB or JetStream.

However, if future workloads demand **high write throughput** or **large-scale event streaming**, **EventStoreDB** would be the next contender. It offers per-stream indexes, optimistic concurrency and real-time subscriptions out of the box【640399355648082†L148-L176】. For purely streaming workloads (e.g., IoT sensor feeds), **NATS JetStream** provides low latency and durable message replay【129778631843866†L46-L55】, but it would require additional operational effort and doesn’t natively provide per-stream versioning.

In summary, **start with a PostgreSQL-based event store** and revisit specialized solutions if scaling requirements outgrow the relational approach. This path aligns with Janus’ modular, offline-first design and minimizes complexity while preserving future flexibility.