Conceptual slots requiring research for the Janus backend

This document enumerates the **conceptual slots** that remain open in the Janus backend architecture. Each slot is described at a high level: its purpose in the system, the key requirements and design challenges (especially around offline-first operation and security), and the types of technologies that may fit. These descriptions do **not** select vendors but instead outline the conceptual space so that further research can be done independently for each slot without blocking other decisions.

1. Document store for semi-structured/unstructured data

Purpose. A generative-AI platform often generates large JSON documents, transcripts, prompts and responses, and other semi-structured files. Using a relational database for these large blobs can be inefficient because rows are designed for uniform, small records; when documents grow, relational engines store them off-row and incur serialization overhead【535312245287 25†L0-L47】. A document-oriented database (e.g., MongoDB, CouchDB) is optimised for storing JSON or binary documents that range from kilobytes to megabytes【535312245287 25†L18-L33】.

Requirements. The document store should:

- Support **hybrid local + cloud** operation so that the system can run completely offline but also scale or replicate data to the cloud when needed. The hybrid storage strategy described for object storage applies here: recent or frequently accessed documents should be available locally for low-latency access, while older or archival data can be synchronised to cloud storage for durability【535312245287 25†L54-L133】.
- Provide **fine-grained access control** and robust **encryption** to secure AI-generated content and personal data. Because generative AI systems can store sensitive prompts, conversations or knowledge graphs, the document store must offer server-side encryption and integrate with the secret-management layer.
- Offer a **Python-friendly driver** for integration with the agent orchestrators (LangChain, AutoGen) and support transactions or consistency models appropriate to the use case (e.g., eventual or strong consistency).

Security considerations. In addition to encryption at rest and in transit, the system should separate user-facing data (e.g., chat logs) from internal system logs and restrict access via roles. Robust auditing and retention policies should be enforced to avoid leakage of sensitive content.

2. Event store / audit log

Purpose. Janus plans to record an **immutable history of agent actions** to facilitate accountability, replay of agent workflows, and debugging【556566689722769†L59-L76】. An event store (sometimes called an event log or audit log) appends events describing each significant action or state change, ensuring no data is lost and enabling time-travel or state reconstruction.

Requirements.

- **Append-only and durable.** The store must guarantee that once an event is recorded it cannot be modified or deleted. Durability is critical because events may be used for compliance or to reconstruct agent state.
- **Offline-first.** The store should operate fully on local hardware and support replicating or exporting logs for disaster recovery or centralised analytics when connectivity is available.
- **Query-ability.** To support monitoring and debugging, events should be indexed by timestamp and key attributes (e.g., agent id, task id) to enable efficient queries and summarisation.

Possible technologies. Options include simple append-only tables in

PostgreSQL, log-structured stores (like Apache Kafka in a self-hosted mode) or file-based event journals integrated with the backup strategy. Research must weigh the overhead of running a message broker vs. a simpler local log and the need for eventual integration with streaming analytics.

Security considerations. Since audit logs may contain sensitive system commands or data, encryption and strict access control are required. Integration with a secret store is essential to protect credentials used for replication or export.

3. Time-series database

Purpose. Monitoring agent performance, resource usage and environmental metrics (e.g., CPU, memory, GPU temperature) requires storing and querying **time-series data**. A time-series database (TSDB) is optimized for chronological data and supports downsampling, retention policies and fast aggregations.

Requirements. The TSDB should:

- Run efficiently on a single device yet scale to clusters; support offline operation and local storage.
- Integrate with observability tools such as **Prometheus** and **Grafana**; data collection may use exporters or sidecars.
- Support data retention policies to purge old metrics while retaining high-resolution recent data.

Security considerations. The TSDB must secure access to metrics, particularly if those metrics could leak usage patterns or system weaknesses. TLS and authentication should be enforced, and any remote access should be limited to trusted monitoring agents.

4. Embedding generator (vectorizer) selection

Purpose. Generative AI agents need to embed documents, code snippets and messages into vector representations for similarity search and retrieval-augmented generation (RAG). The embedding generator must be self-hostable to preserve privacy and operate without internet connectivity.

Requirements.

- **Local model support.** Models such as NV-Embed-v2, Qwen3, Jina Embeddings and BGE-M3 have been highlighted in external research for strong retrieval performance and multilingual support【540803978116501†L48-L71】【540803978116501†L89-L110】. The chosen model should run on available hardware (CPU or GPU) and offer a permissive licence for commercial use; some models (e.g., NV-Embed) are CC-BY-NC and may not be suitable.
- **Integration with Qdrant and RAG frameworks.** The output embeddings must be compatible with Qdrant (the locked-in vector store) and the RAG retriever layer. This typically means producing fixed-dimensional vectors in a format accepted by Qdrant’s API.
- **Performance and resource usage.** Because Janus may run on edge devices, the embedding model must be efficient and support quantisation or CPU inference to minimise resource consumption. Support for batching and streaming can further reduce memory overhead.

Security considerations. Running models locally reduces the risk of leaking embeddings to external services. Care must be taken to keep model files secure and to verify that open-source models are sourced from reputable repositories to avoid supply-chain attacks.

5. Local language-model (LLM) execution framework

Purpose. To run large language models offline, Janus needs a local LLM framework or engine. Candidate frameworks include **Ollama**, **llama.cpp** and **vLLM**. External research notes that each has different strengths: Ollama emphasises ease-of-use with a built-in REST API; llama.cpp offers hardware-optimised inference with quantisation; and vLLM provides high throughput for large models but assumes GPU availability【250089150898544†L36-L93】【250089150898544†L96-L160】.

Requirements.

- **Hardware compatibility.** The framework must run on consumer GPUs (e.g., RTX 3060/3080) and degrade gracefully to CPU inference when GPUs are unavailable. Rootless operation would enhance security by avoiding privileged containers.
- **Model management.** Support downloading and loading models in open formats (GGUF/LLAMA) and managing versions. For a self-hosted system, the framework should not require internet access once models are downloaded.
- **API integration.** Provide a simple API (REST, gRPC or Python) to integrate with the agent orchestrator. Support streaming responses and concurrency (multiple agents querying the model) is desirable.

Security considerations. Running models locally mitigates the risk of leaking prompts or user data. The framework should isolate model execution from the host (e.g., through containerisation or user namespaces) and avoid executing untrusted code. Signed model files and checksums are advisable to detect tampering.

6. Retrieval-Augmented Generation (RAG) retriever layer

Purpose. Janus relies on **hybrid retrieval**—combining vector similarity search from Qdrant with relationship traversal in the Neo4j knowledge graph—to supply context for the LLM. The retriever layer coordinates these queries and summarises results before feeding them to the model.

Requirements.

- **Direct Qdrant + Neo4j integration.** Tools like **LlamaIndex** provide out-of-the-box connectors for Qdrant and Neo4j and support building composite indices【894240467193786†L26-L45】. They also provide summarisation and hierarchical context features【894240467193786†L48-L55】. Alternatives like **Haystack** support Qdrant but offer less seamless graph integration【894240467193786†L118-L141】.
- **Support for agent workflows.** The retriever must handle multi-step queries (e.g., search → reason → search) and streaming to the LLM. The ability to integrate with LangChain or AutoGen tools as a “retrieval tool” is important.
- **Python ecosystem.** A strong Python API is essential for integration with the rest of the stack.

Security considerations. The retriever layer should enforce access controls on underlying stores (Qdrant and Neo4j) and sanitise user queries to prevent injection attacks. When summarising or aggregating sensitive data, ensure that only authorised agents or users see private context.

7. Caching layer

Purpose. A caching system stores transient or computed data (e.g., session state, resumable job states, vector cache) to reduce latency and relieve pressure on primary databases. The Janus stack emphasises durability over raw speed and requires the cache to support both in-memory operations and persistence【588244271614106†L6-L20】.

Requirements.

- **Self-hosted and distributed.** The cache must run across multiple machines and support replication or clustering to scale horizontally【588244271614106†L6-L11】.
- **Durable persistence.** Redis, for example, offers snapshots (RDB) and append-only files (AOF) to ensure data survives crashes【588244271614106†L41-L51】. The system should allow tuning persistence per use case (e.g., TTL keys for sessions vs. durable keys for job states)【588244271614106†L15-L21】.
- **Python integration and observability.** The cache should integrate with Python libraries and support monitoring via Prometheus exporters【588244271614106†L27-L32】.

Alternative options. Emerging alternatives like **Dragonfly** or **KeyDB** provide multi-threaded performance and built-in persistence. Research must compare them against Redis in terms of durability, performance, ease of administration and licensing.

Security considerations. Since the cache may store sensitive tokens or intermediate data, encryption at rest and in transit, as well as authentication and access control, are required. Lua scripting (used in Redis) must be sandboxed to prevent arbitrary code execution.

8. Task queue / scheduling & throttling system

Purpose. Agents will offload background tasks (document ingestion, parsing, AI workflows) to a job queue. The system should support **mixed worker types**, scheduling of periodic tasks, and high reliability【446268711490948†L6-L19】.

Requirements.

- **Scalability and heterogeneity.** The queue must run workers on different machines and allow tasks to be routed to specific worker types (CPU vs. GPU)【446268711490948†L6-L19】.
- **Built-in scheduling.** Support cron-like periodic jobs and delayed execution without complex custom code【446268711490948†L12-L14】.
- **Integration with Python and existing stack.** The solution should work with RabbitMQ (the chosen message broker) or alternative brokers and integrate with Dramatiq, Celery or other Python libraries.

Research space. Possible options include **Celery**, **Dramatiq** (already used), **RQ** and **ARQ**. Celery offers comprehensive features but is heavier to configure【446268711490948†L23-L56】. RQ is lightweight but less durable and lacks advanced features【446268711490948†L96-L149】. Research should compare their reliability, scheduling capabilities and maintenance overhead.

Security considerations. Ensure tasks are authenticated and validated to prevent arbitrary code execution via the queue. Monitor and limit queue lengths and worker concurrency to avoid denial-of-service conditions. Use encrypted connections (TLS) between brokers and workers.

9. Observability: monitoring and logging

Purpose. Observability is critical to run a self-hosted system safely. It includes **monitoring resource usage**, **centralised logging** and **tracing** of agent tasks. Without good observability, debugging edge deployments or recovering from failures becomes very difficult.

9a. Metrics collection and resource monitoring

A metrics system (Prometheus plus exporters) should collect CPU, memory, GPU and network statistics from each service. The metrics data will feed into the

time-series database (Slot 3) for storage and be visualised via Grafana or similar dashboards.

9b. Container monitoring and orchestration

The current stack uses Podman/Quadlet and will migrate to Kubernetes. Podman's integration with systemd allows rootless operation, auto-restart, and centralised logging to journald【870520200600953†L87-L129】. However, it lacks multi-node awareness【870520200600953†L132-L160】, so external tools like Nomad or Kubernetes will be needed for clustering. Research should evaluate whether to adopt a lightweight orchestrator (e.g., Nomad) or move directly to K3s/Kubernetes. Observability should include container health checks, resource limits, and auto-updates with rollback support【870520200600953†L86-L124】.

Security considerations. Running containers in rootless mode improves isolation【870520200600953†L104-L111】. Logs should be centralised (e.g., using journald or Loki) and protected against tampering. Secret values must not appear in logs; configure log scrubbing and role-based access to dashboards.

10. Object storage

Purpose. The platform needs a **self-hosted object store** to hold large files (documents, AI model artifacts, backups) and optionally synchronise to cloud storage. Hybrid local-first storage ensures low latency and data sovereignty while providing scalability and durability【535312245287 25†L54-L137】.

Requirements.

- **Versatility and durability.** The storage must handle all file types and provide erasure coding or replication to protect against disk failures【227469825217964†L60-L68】.
- **Hybrid local + remote operation.** It should allow local access while supporting replication or tiering to cloud providers (e.g., Amazon S3) for redundancy【227469825217964†L25-L33】【227469825217964†L69-L78】.
- **Security and access control.** The store must provide TLS, server-side encryption and IAM-like policies to control access【227469825217964†L21-L33】【227469825217964†L139-L167】.
- **Developer tooling.** S3-compatible APIs and Python SDK/CLI are essential【227469825217964†L33-L35】.

Candidate technologies. Research has identified options such as **MinIO**, **Ceph (RADOS Gateway)**, **SeaweedFS**, **OpenIO**, **Zenko** and **Garage**【227469825217964†L60-L109】. Each differs in complexity and features. MinIO offers simplicity and speed, Ceph provides high scalability at the cost of complexity, and Garage targets small clusters with decentralised DHT storage. These will need detailed evaluation during the vendor-selection phase.

Security considerations. Server-side encryption keys must be managed by a secret-management system (see Slot 12). Access policies must be defined per bucket and per service. Audit logging should track all object access.

11. Backup and disaster-recovery (DR) system

Purpose. Given the offline nature and eventual scaling to clusters, Janus needs a robust **backup and disaster-recovery** plan. Backups must cover all core services (PostgreSQL, Neo4j, Qdrant, MinIO, Redis, RabbitMQ) and be tested regularly【839148278490595†L0-L29】.

Requirements.

- **Multi-data-source support.** Automated scheduled backups for each component using appropriate methods (SQL dumps, snapshot APIs, file copies)【839148278490595†L5-L15】.
- **Encryption and offline storage.** Backups must be encrypted and stored locally; optional sync to external media or object storage is allowed【839148278490595†L12-L18】.
- **Automated verification.** Tools should enable periodic test restores or integrity checks【839148278490595†L25-L31】.
- **Declarative configuration and low maintenance.** Backup jobs should be defined in config files (YAML or similar) and integrate with containerised services【839148278490595†L32-L45】.
- **Migration to Kubernetes.** The chosen solution should have a path to integrate with Kubernetes backups (e.g., Velero) in later phases【839148278490595†L47-L54】.

Candidate tools. Options include deduplicating backup tools like **Restic**, **BorgBackup/Borgmatic** (lightweight CLI with encryption and deduplication)【839148278490595†L55-L74】; backup orchestrators like **SHIELD**; container-native tools like **docker-volume-backup**; and Kubernetes-focused tools (Velero, Stash). Research should compare them on automation, restore verification and complexity.

Security considerations. Backups contain all data and must be encrypted (client-side) before storage; keys must be managed securely. Access to backups should be restricted and audited. Disaster-recovery procedures should be documented and tested.

12. Secret management and configuration

Purpose. Managing credentials, API keys and encryption keys securely is essential for protecting services and data. Janus must integrate a **secret-management solution** that works offline, supports local encryption, and scales to a cluster.

Requirements.

- **Self-hosted secret store.** Options like **HashiCorp Vault**, **Keycloak** (for IAM) and **SOPS** (sealed secret files) provide ways to store and distribute secrets. The solution should operate offline and support rootless containers.
- **Integration with container runtimes.** Secrets must be injected into containers securely (e.g., via environment variables or mounted files) without embedding them in images. For Kubernetes later, integration with K8s secrets/CSI drivers is needed.
- **Encryption and access control.** The secret store should encrypt data at rest using strong ciphers and enforce role-based access control. Audit logs of secret access are required.

Security considerations. The secret management system itself must be hardened and regularly updated. Root tokens or master keys should be stored offline or under multi-party control. Secrets should be rotated automatically where possible.

13. Identity and access management (IAM) / API gateway

Purpose. As Janus evolves to serve multiple users or expose APIs to external clients, an **IAM layer** will be required. This includes user authentication (possibly multi-factor), authorisation and API rate-limiting.

Requirements.

- **Self-hosted identity provider.** Potential candidates include **Keycloak**, **Authentik**, or **Authelia**, which provide OAuth2/OIDC, SAML and support offline operation. The IAM should manage user accounts, roles, and permissions and issue tokens for service-to-service communication.
- **API gateway.** A gateway like **Kong** (self-hosted) or **Traefik** can provide rate-limiting, TLS termination and routing while integrating with the identity provider.
- **Agent-level access control.** In a multi-agent system, each agent may require specific scopes; the IAM should issue tokens with the minimum required privileges.

Security considerations. The IAM must enforce strong password policies, multi-factor authentication and secure session management. The API gateway should filter requests for injection attacks and provide centralised logging of all API calls.

14. Container registry and build pipeline

Purpose. To run containers offline, Janus needs a **private container registry** to store and distribute images for Postgres, Neo4j, custom services and AI models. A build pipeline (e.g., using **GitHub Actions** or a self-hosted CI like **Drone** or **Woodpecker**) will build, scan and sign images before pushing them to the registry.

Requirements.

- **Self-hosted registry.** Options include **Harbor**, **Docker Registry**, **Sonatype Nexus** or **GitHub Container Registry** (if connectivity exists). The registry must run offline and support role-based access and image scanning.
- **Build automation.** The pipeline should run tests, vulnerability scans and produce reproducible images. For offline builds, caching dependencies (e.g., Python wheels) is essential.
- **Image signing and verification.** Use tools like **Cosign** to sign images; the runtime (Podman/Kubernetes) should verify signatures to prevent supply-chain attacks.

Security considerations. The registry must be secured with TLS and authentication; only trusted builders should push images. Vulnerability scanning should be part of the build pipeline to detect insecure dependencies.

15. Scheduling & resource throttling for AI agents

Purpose. Running multiple AI agents concurrently can overwhelm hardware (CPU, GPU, memory). A **scheduling and throttling** component allocates resources and prevents one agent from starving others. This is distinct from the task queue; it deals with per-agent concurrency and GPU scheduling.

Requirements.

- **Awareness of hardware limits.** The scheduler should know the number of CPU cores, GPU memory and allocate tasks accordingly. In a Kubernetes environment, this could be handled by resource requests/limits and device-plugin scheduling.
- **Policy-based throttling.** Support configurable policies such as maximum concurrent GPU tasks or priority levels for critical agents vs. background jobs.
- **Integration with the job queue.** The scheduler and queue should coordinate: tasks remain queued until resources are available. GPU-enabled tasks might be routed to dedicated workers.

Security considerations. Prevent malicious or runaway agents from monopolising resources. Ensure that GPU memory is zeroed between jobs to prevent data leakage across tasks.

Conclusion

This overview defines the conceptual slots in the Janus backend that remain open for research. Each slot includes key requirements and security considerations anchored in the Janus principles of offline-first operation, self-hostability, resilience and modularity【556566689722769†L14-L41】. With this high-level framework in place, you can now conduct focused vendor evaluations for each slot in parallel, confident that these slots are well scoped and that their selection will not inadvertently constrain other components.