ChatGPT

Janus Backend Architecture – Onboarding Brief

Project Vision and Guiding Principles

Long-Term Vision: The Janus project's mission is to create a **resilient, self-hostable agent orchestration platform** that can operate entirely **off-grid** on local hardware, yet seamlessly scale out to cloud infrastructure when needed. In other words, Janus should run on a single rugged device without internet, but also expand horizontally or burst into managed cloud services under heavier loads. This dual capability is where the name "Janus" (the two-faced Roman god) is fitting – the system looks both inward (offline, autonomous operation) and outward (scalable, networked operation).

Guiding Principles:

- **Offline-First & Self-Sufficiency:** Janus is designed to function without external dependencies. All core components are open-source and deployable on-premise, ensuring that even with no internet connectivity, the system remains fully operational. This principle supports privacy (sensitive data stays local) and reliability in remote or secure environments.

- **Resilience and Fault Tolerance:** Robustness is prioritized over convenience. Each chosen component has proven stability and supports data durability or fail-safes. The architecture avoids single points of failure where possible. Notably, all core components have available ARM64 builds and documented offline installation paths – the stack can literally run on a **solar-powered mini-PC** in the field. Power loss, network interruptions, or component crashes should not result in data loss or system-wide failure; messages queue up, services recover gracefully, and the agent persists.

- **Modularity & "Best-of-Breed" Components:** The system is composed of specialized modules (relational DB, graph DB, vector index, etc.), each selected for excellence in its role. This contrasts with an all-in-one monolith. By using loosely coupled components, Janus can leverage the best tools available for each function (storage, computation, messaging) and swap them out if requirements change. This modularity also simplifies scaling and troubleshooting – each service can be tuned or scaled independently.

- **Scalability and Flexibility:** While offline solo deployment is a must, Janus is built with scaling in mind. The same architecture should handle a personal knowledge base on a Raspberry Pi or a cluster of agents performing enterprise-grade workloads. This means using components that support horizontal scaling and clustering (e.g. RabbitMQ for messaging, PostgreSQL which can cluster or replicate, etc.). **Elasticity** is a goal – the ability to "burst" to cloud might mean spinning up managed versions of these components temporarily and then syncing back on-prem. The design decisions weigh scalability so that adding more agents or more data won't necessitate a complete redesign.

- **Consistency and Data Integrity:** Agents will be handling critical data and actions, so the architecture ensures strong consistency where needed. For example, PostgreSQL offers ACID

transactions for important state (preventing corruption or conflicts), and other stores are used in roles appropriate to their consistency models (Neo4j for graph consistency, etc.). Audit logs and an event store (planned) will provide an immutable history of actions for accountability. Data integrity is also maintained by clear separation of concerns – ephemeral vs persistent data are stored in different systems to avoid accidental loss or bloat.

- **Openness and Community Alignment:** All chosen technologies are open-source or have open-source core editions. This not only aligns with the self-hostability goal but also means Janus can evolve alongside a community. We favor technologies with active communities and rich ecosystems (for plugins, integrations, and community support). Where a chosen component has licensing limitations (e.g., Neo4j's community vs enterprise license), we have acknowledged those and planned around them (using community edition within its limits, and keeping an eye on alternatives if our use outgrows those limits).

- **Security and Privacy by Design:** Running off-grid inherently provides a security advantage (no external data leaks), but we also plan for secure operation in networked mode. This means using secure communication (e.g., TLS where applicable), managing secrets properly (credentials for databases, tokens for agents), and planning for an identity and access management layer when multi-user capabilities or external API access are introduced. The architecture should be able to enforce authentication and authorization, so that as Janus exposes services (like a future unified API or dashboard), only permitted entities can interact with the system. Additionally, data governance is considered – e.g., an internal knowledge graph might contain sensitive info, so access to it should be controllable per agent or user role in future.

- **Performance within Constraints:** Given the ambition to run on limited hardware, the stack emphasizes lightweight, efficient components. For instance, a Rust-based vector DB was chosen for speed with low overhead, and we avoid overly heavy solutions (no huge Hadoop cluster requirements or overly memory-hungry services by default). Where possible, we choose asynchronous, event-driven approaches to maximize throughput on limited resources. This also means using caching and optimized data access patterns to reduce unnecessary load (e.g., using Redis to cache expensive computations). The guiding philosophy is to achieve acceptable performance on a single machine, which in turn means excellent performance on a stronger multi-node setup.

These principles guided every architectural decision. Together, they ensure that Janus's backend not only meets current needs but provides a solid, extensible foundation for future development by you, the incoming research agent. In the sections below, we outline the concrete components locked in under these principles, explain why they were chosen (and what was left by the wayside), and how they interconnect to realize the above vision.

Core Architecture Components (Locked-In Stack)

Each core component was selected after extensive comparative research, filling a specific role in the system. Below we summarize each component's purpose, why it was chosen, and note alternative technologies that were considered and rejected for continuity. This provides context on how we arrived at the current stack.

Relational Database – PostgreSQL (with SQLAlchemy & Alembic)

Purpose & Role: PostgreSQL serves as the primary **relational database** for Janus. It stores canonical structured data: agent configurations, user accounts/permissions (if any), audit logs of agent actions, and other structured records that require strong consistency (transactions) and schema. It's also the backend for certain components like authentication data and any tabular knowledge the agents maintain. We interface with Postgres through SQLAlchemy (an ORM) for convenient data modeling in Python, and Alembic is used for migrations to evolve the database schema safely over time.

Why PostgreSQL was chosen: PostgreSQL is a battle-tested, enterprise-grade open-source RDBMS with full ACID compliance. It has a rich feature set and extension ecosystem which we may leverage (JSONB support, GIS extensions, full-text search, etc.), giving us flexibility for future needs. Critically, Postgres can run **offline on virtually any hardware** (from a server to a Raspberry Pi) and still handle concurrent loads gracefully. It's known for reliability and data integrity – exactly what we need for core records. Additionally, the Python async support and tooling around Postgres are excellent; for example, async drivers and compatibility with frameworks ensured that we could integrate it without hassle. By using SQLAlchemy, we keep the option to switch databases in theory, but PostgreSQL's capabilities (advanced queries, indexing options, materialized views, etc.) make it a solid long-term choice. We also valued its **mature ecosystem** of tools for backup, replication, and monitoring.

Alternatives considered: We evaluated a few other databases for this role:

- *MySQL / MariaDB:* Like Postgres, MySQL is a popular open-source RDBMS. We decided against it mainly because PostgreSQL offered more advanced features that could be relevant (e.g., richer JSON support for semi-structured data, window functions, CTEs) and a more permissible license for embedding (Postgres is MIT-like). The Postgres community and extensions (such as PostGIS if needed) were stronger advantages for our use case. MySQL was certainly capable, but provided no clear benefit over Postgres in our context, and we preferred to standardize on one primary SQL database – Postgres's track record for correctness (ACID compliance, transactional DDL, etc.) gave us confidence.

- *SQLite:* We considered SQLite for a truly lightweight option (it requires no server), which could work for a single-machine deployment. However, SQLite cannot handle concurrent writes from multiple processes well and lacks the client-server architecture needed for scaling. Given Janus is meant to scale and potentially have multiple agent processes or even multiple nodes, SQLite was not sufficient beyond simple prototyping. PostgreSQL, by contrast, can start small (it can run in a low-memory mode on a Pi) but then scale up to large deployments, making it a more future-proof choice.

- *NoSQL or NewSQL stores:* For the core data, we briefly asked if a NoSQL document store could replace the relational DB. We concluded that while we will have a document store separately, the relational model is crucial for structured, interrelated data that requires transactions (e.g., updating an agent's config and logging that action in one go). NewSQL options (like CockroachDB or MariaDB XtraDB cluster) were overkill for now and would add complexity that isn't needed until possibly much later (if we needed multi-node active-active SQL, which we don't at this stage).

In summary, **PostgreSQL was locked in** as the structured data backbone for its reliability, rich features, and alignment with our off-grid and scalability requirements.

Graph Database – Neo4j (Community Edition)

Purpose & Role: Neo4j is our **graph database** dedicated to long-term semantic memory – essentially the knowledge graph for Janus. It stores entities and relationships that the agents learn or use, enabling complex queries over those relationships. For example, if the AI agent builds an ontology of concepts or a network of how certain ideas connect, that lives in Neo4j. This is separate from the relational data to allow efficient graph traversal queries (pathfinding, subgraph matches) which are not feasible in a SQL schema. We expect to use Neo4j to answer questions like "How is concept X related to Y?" or "Retrieve all steps in the chain of reasoning linking A to B" in a way that's much more direct than a relational approach.

Why Neo4j was chosen: Neo4j is the leading native graph database and has a powerful and expressive query language (Cypher) that our agents can utilize for reasoning. It's known for its performance on traversal-heavy workloads and for being developer-friendly when modeling knowledge graphs. We chose Neo4j Community Edition specifically because it's open-source (GPL licensed) and can be run offline without licensing costs. It integrates well with Python – there are official drivers and ORMs (like Py2Neo or the Neo4j Python driver), which means our agents can query the graph easily. The decision was also influenced by Neo4j's maturity and documentation; as a widely adopted graph DB, it has many resources for troubleshooting and examples for usage (useful for an AI agent developer coming onboard). In short, Neo4j offered a **robust, ready-to-use solution for graph data** with features like indexing, ACID transactions on the graph, and an active ecosystem for things like graph algorithms. We were confident it could handle our knowledge graph needs out of the box. (Community Edition limitations like lack of clustering were acceptable, since initially we run on a single node. We note that if we ever *did* need multi-node or high availability, an upgrade or alternative might be required – see Open Questions.)

Alternatives considered: We did a comparative analysis of other graph stores:

- *JanusGraph:* An open-source, distributed graph database (born from the Titan project) that can run on big data backends (Cassandra, Scylla, HBase, etc.). While JanusGraph is very scalable and **free for enterprise use (Apache 2 license)**, it introduces a lot of operational complexity (you need to run and maintain those backends). It shines with huge graphs, but our initial scope is manageable on a single node. Running JanusGraph on a single machine is possible (using an embedded BerkeleyDB or similar), but then we'd lose some benefits of its distributed nature. Given our **off-grid, modest-hardware target**, JanusGraph felt like over-engineering; Neo4j was far simpler to deploy and use for our size, and its performance on single-node workloads is hard to beat. We kept JanusGraph in mind mainly if we hit Neo4j's community edition limits in the future, but that was deemed unlikely in the near term.

- *ArangoDB:* A multi-model database that includes a graph engine alongside document and key-value models. ArangoDB could have, in theory, let us use one database for multiple purposes (it can do graph and also serve as a document store). We considered it to potentially reduce the number of different systems. However, we determined that Arango's graph capability, while good, wasn't as optimized or feature-rich in graph querying as Neo4j's (Cypher vs Arango's AQL for complex graph traversals). Also, using Arango in multi-model mode might simplify architecture diagrams but can complicate performance tuning (we'd be using one system for very different workloads). We preferred **separation of concerns** – a dedicated graph DB that excels at graph operations. Additionally, ArangoDB's footprint and memory usage tend to be higher when all features are enabled. We decided to leverage ArangoDB possibly for a document store in the future (it was listed

as one candidate for that slot), but **not to replace Neo4j for the semantic graph**, where Neo4j had the clear edge in query power and community support.

- *TigerGraph, Dgraph, Memgraph:* We briefly looked at other graph databases. TigerGraph is high-performance but a commercial product (with limited community edition capabilities) – not fitting our open self-host criteria. Dgraph is an open-source graph DB with a GraphQL interface; it's interesting, but the project had some turbulence (company behind it pivoted), and it's also more of a distributed system by design, which we didn't need yet. Memgraph is a newer in-memory graph DB that also uses Cypher; it's partially open-source but with closed-source core components in the paid version. Memgraph's focus on real-time in-memory operation made us concerned about persistence and running on low-power devices (it's also more niche). Ultimately, Neo4j's **market leader position** and developer familiarity made it the comfortable choice.

- *No Graph DB (just use relational or other stores):* For completeness, we considered if we could model the necessary relationships in PostgreSQL or in the vector store. Simple relationships can indeed be handled in a relational model or by embedding graphs in other forms, but for complex relationships and quick traversal (like "find all nodes connected within 4 hops"), those approaches become unwieldy and slow. The need for flexible queries (potentially generated by an AI agent) was a strong argument that a purpose-built graph database was the right tool.

We acknowledge one consideration: **Neo4j licensing.** Community Edition is GPL and free to use, but if our project ever required features like clustering, we'd either need a Neo4j Enterprise license or to migrate to an alternative. For now, we've **"parked" the licensing concern**, as Community Edition meets our offline R&D needs and has no blockers for local deployment. We have documented this for the future (see Open Questions), but it does not affect current functionality.

Vector Database – Qdrant

Purpose & Role: Qdrant is our chosen **vector database**, which is used for similarity search on embeddings as part of Retrieval-Augmented Generation (RAG). Whenever the agent needs to find relevant pieces of unstructured information (documents, knowledge snippets, etc.) based on semantic similarity, those items are stored as high-dimensional vectors in Qdrant. For example, if Janus ingests a set of articles or documentation, we generate embeddings for chunks of that text and store them in Qdrant; later, given a query, we retrieve the closest matches by vector distance. The vector DB is thus a key component for the agent's long-term memory of unstructured text, enabling it to **"remember"** and retrieve information efficiently that doesn't fit in the graph or relational DB.

Why Qdrant was chosen: Qdrant stood out as a **lightweight, high-performance, and local-first** vector store. It's written in Rust, which gives it strong performance with low resource usage – important for running on modest hardware. Qdrant is open-source and designed to be developer-friendly, with features like a simple REST API and support for payload filters (meaning we can store metadata with vectors and filter results by criteria, useful for segmented knowledge). Importantly, Qdrant supports running in embedded or offline modes and even has optional GPU acceleration for similarity search, which we could leverage if the hardware allows. During our research, we found that Qdrant's roadmap and community were very aligned with edge and embedded deployments – in fact, a recent update introduced a version optimized for *embedded devices*, emphasizing local operation without heavy server overhead. All this fits our off-grid requirement well.

Additionally, Qdrant's performance benchmarks showed competitive or superior results for throughput and latency in similarity search tasks. It provides vector indexing strategies and quantization options to handle large volumes efficiently. For our initial use (which might be a few thousand to millions of embeddings), Qdrant can handle that on a single node with ease. Another factor was integration: there are Python client libraries, and LangChain has connectors for Qdrant, making it easier to plug into our agent loops. The **"live filter"** capability (filtering search by metadata like tags) is very useful when the agent wants to restrict search to certain contexts (e.g., "only search among technical docs, not personal notes").

Alternatives considered: We looked at both open-source and managed vector DB options:

- *Weaviate:* An open-source vector search engine with a lot of features (and a managed cloud option). Weaviate supports hybrid search (combining keyword and vector) and has a GraphQL-based query API. While attractive, Weaviate has a relatively larger footprint – it's written in Go, requires more memory for its indices, and internally can use additional services (it can offload vector storage to an ext. DB if needed). For an **edge deployment**, Weaviate felt heavier than Qdrant. We also didn't immediately need some of its advanced modules (like built-in classification or Q&A extensions). Weaviate could have been overkill, and its baseline resource usage was a concern given our target device scenarios.

- *Milvus:* A popular open-source vector database (from Zilliz). Milvus is highly scalable and plugin-oriented (it has a distributed architecture relying on etcd, and can use different storage engines). We considered Milvus especially if we envisioned extremely large vector sets. However, Milvus's architecture (multiple services, dependency on etcd, heavy memory usage for query nodes) was not ideal for a single small machine. It's better suited if we needed a cluster handling billions of vectors. Our conclusion was that Milvus introduced unnecessary complexity for Janus's scale, and that Qdrant could handle our needs more simply. Milvus might become a candidate if we ever needed to integrate with its ecosystem or handle truly massive data, but not for now.

- *Postgres with pgvector:* Rather than a separate vector DB, one idea was to use PostgreSQL itself to store embeddings using the pgvector extension (which allows vector similarity queries in SQL). This would simplify the stack by not having another service. We debated this option; it's quite convenient (fewer moving parts). However, performance tests generally show dedicated vector databases outperform a general SQL DB for similarity search, especially as the number of vectors grows or when doing approximate nearest neighbor search with specific indexes. Also, offloading this to Qdrant means we can scale vector search independently and use GPU optimizations. Using Postgres for everything could have created a performance bottleneck and made tuning difficult (mixing OLTP workloads with large vector searches in one database). So while pgvector remains a handy tool (and we may use it for quick hacks or small-scale similarity inside Postgres), **we opted for a specialized component (Qdrant) to do the heavy lifting** for RAG.

- *Chroma or other lightweight stores:* We also looked at ultra-light alternatives like Chroma (an open-source Python-based vector store) and FAISS (Facebook's library for similarity search). Chroma could be embedded in our application, but it's more of an in-memory store with persistence options, and at the time of evaluation it was still maturing and aimed at small to medium use cases. FAISS is a library, not a full DB server – we could integrate FAISS directly for vector search, but we'd have to build surrounding features (network service, metadata filtering, etc.) ourselves. Given that Qdrant offered a ready-made solution with those features, we preferred not to reinvent that wheel.

- *Managed SaaS (e.g., Pinecone, Vespa Cloud):* We ruled these out because they violate the off-grid requirement. Pinecone, for instance, is a powerful hosted vector DB, but using it would tether Janus to an internet connection and external dependency. Part of our guiding principles is to avoid vendor lock-in and ensure the system can run in isolation, so any managed service was a non-starter for core functionality.

In summary, **Qdrant was selected** for being resource-efficient, self-hostable, and feature-complete for our needs. It aligns with the "local-first" ethos (even targeting edge devices explicitly) and has the performance to support Janus's semantic search capabilities. We will monitor its development and scalability as usage grows, but it provides an excellent starting point.

Message Broker – RabbitMQ

Purpose & Role: RabbitMQ is employed as the **message broker** in Janus. It's the backbone of our asynchronous communication, enabling different parts of the system (and different agent processes) to exchange messages reliably. Whenever an agent schedules a background task or two components need to work in a decoupled manner, those messages go through RabbitMQ. Concretely, it's used in conjunction with our task queue (Dramatiq) to queue jobs, but could also be used for other pub/sub patterns if needed. The broker ensures that work is **durably stored and delivered** from producers to consumers, handling retries, acknowledgements, and load balancing between consumers.

Why RabbitMQ was chosen: RabbitMQ is a proven, stable choice for messaging with a relatively small footprint. It supports many-to-many communication via topics and queues, persistent messaging to disk, and has robust delivery guarantees (acknowledgments, message persistence, etc.). We favored RabbitMQ because it excels in scenarios where you want strong reliability and control over how messages are routed and processed. It's also **lightweight enough to run on a low-power device** (there are reports of RabbitMQ on Raspberry Pi handling decent loads, and it's included in many IoT stacks). Importantly, RabbitMQ plays nicely with Python through standard libraries (pika or others) and is directly supported by Dramatiq (our task library) as a broker option. We can configure RabbitMQ for priority queues, delayed message plugins, or other advanced patterns if needed later.

Another deciding factor was RabbitMQ's ability to handle **back-pressure**: if consumers (workers) are slow or offline, RabbitMQ will queue messages and not overwhelm them when they come online. It also allows for fair dispatch (round-robin to workers) and other patterns that we anticipated needing. Essentially, RabbitMQ gave us a **durable, managed queue system out of the box**, which is preferable to building something ad hoc or using a less feature-rich system. We knew from our research that using a purpose-built MQ would avoid pitfalls like losing messages or overrunning memory – RabbitMQ has been field-tested in those aspects.

Alternatives considered:

- *Using Redis for messaging (Redis Streams):* A notable alternative was to avoid a separate broker and instead use Redis (which we already need for caching) to also handle streams of messages (Redis Streams feature). This could simplify the stack by not having an extra service. We analyzed this trade-off: Redis Streams do allow persistent message logs and consumer groups (kind of like a lightweight Kafka). However, Redis is single-threaded for most operations and does not apply back-pressure in the same way – if a consumer falls behind, Redis will keep data in memory which can grow, and it

doesn't naturally signal producers to slow down. RabbitMQ, on the other hand, can persist to disk and has flow control to handle overload situations. Moreover, Redis Streams would tie our task system to Redis's semantics, potentially complicating the use of Dramatiq (which natively supports Redis but our preference was to separate concerns). Ultimately, **RabbitMQ was chosen over Redis Streams** because it offers stronger delivery guarantees and more robust queue semantics for task distribution. We kept Redis in the architecture but strictly for fast cache/pub-sub, where its in-memory nature is an advantage and its single-thread limitation is acceptable.

- *Apache Kafka:* Kafka is a high-throughput distributed log, often used for event streaming at scale. It ensures durability and can handle enormous message volumes with partitioning. We considered Kafka for a moment, thinking ahead to if we needed a unified log of events or a way to integrate a lot of streaming data (sensor feeds, etc.). However, Kafka has significant overhead – it's JVM-based, requires running Zookeeper (in older versions) or additional coordination, and is not easy to run on a small device or to justify for moderate workloads. Kafka shines in big data pipelines and multi-consumer analytics scenarios, but for Janus's immediate needs (internal task/job queueing and moderate message volumes), it would be **overkill and very resource-intensive**. RabbitMQ is much simpler to configure and maintain for our scale. We decided that if in the future a need arises for high-volume event streaming (e.g., thousands of events per second, long retention), we might introduce Kafka or a similar log system as a new component, but not as a replacement for Rabbit's targeted role in task queuing.

- *Others (ActiveMQ, NATS, ZeroMQ, etc.):* We looked at other messaging systems:

  - **ActiveMQ** (and its newer version, Artemis) is another durable JMS broker similar to RabbitMQ. It wasn't chosen because RabbitMQ and ActiveMQ have comparable features, but RabbitMQ has a bigger community in the context of Python and DevOps, and we had more prior familiarity with it.
  - **NATS** is a very lightweight messaging system focused on simple pub-sub (and a newer NATS Streaming adds durability). NATS is extremely fast but trades off persistence; NATS Streaming (JetStream) can persist but it's a newer approach. We found RabbitMQ's feature set (e.g., routing keys, different exchange types, pluggable auth and management UI) more mature for what we might need. Rabbit also integrates directly with our task library, whereas using NATS would require more custom integration.
  - **ZeroMQ** is a lower-level messaging library (no central broker, peer-to-peer sockets). While very performant, using ZeroMQ would push a lot of complexity onto us to implement reliable delivery and message queuing logic at the application level. It wasn't aligned with our desire for a ready solution with minimal fuss.

In summary, **RabbitMQ was locked in** as the message broker for its reliability, small footprint, and synergy with our Python task system. It acts as the nervous system of Janus, making sure work gets where it needs to go, even in adverse conditions (it will safely hold tasks on disk if workers crash or are busy). This choice strongly supports our resilience and scalability objectives (it's easy to add more worker consumers or even federate RabbitMQ if needed later).

Task Queue & Orchestration – Dramatiq

Purpose & Role: Dramatiq is the **task queue library** used for executing background jobs and scheduling work in Janus. Where RabbitMQ is the messaging infrastructure, Dramatiq is the Python-level framework

that allows us to define “actors” (task functions) and send tasks to be run asynchronously by worker processes. It handles retrials, scheduling (deferred execution or repeated tasks), and result collection for tasks. In practical terms, when the Janus agent needs to perform an operation that shouldn’t block the main loop – such as fetching data from the web, running a long computation, or any I/O-heavy process – that work is packaged as a Dramatiq task and put on RabbitMQ. Worker processes running Dramatiq will consume the task, execute it, and then acknowledge completion. This pattern is essential for enabling **concurrency and parallelism** in the system, so that the AI agent can manage multiple objectives and I/O-bound tasks efficiently.

Why Dramatiq was chosen: We chose Dramatiq after evaluating it against other Python task queues (like Celery). Dramatiq appealed to us for several reasons: it’s relatively **lightweight and simple** to use, with a clear and concise API (using Python decorators to declare tasks). It was designed to be performant and correct from the ground up – for instance, tasks are only acknowledged after completion by default, avoiding issues where a crash could lose a task. Dramatiq integrates smoothly with RabbitMQ (and Redis) and supports **asynchronous (async/await)** task definitions, which is great for modern Python usage (Celery only recently began to embrace async and can be trickier in that regard). Dramatiq also includes helpful middleware out of the box, such as automatic retries, timeouts, and rate limiting, which we knew we’d want. In terms of resource use, Dramatiq has a smaller footprint and a simpler concurrency model compared to Celery. It doesn’t need external dependencies like a results backend unless we choose to use one (we do plan to use Redis as a results backend for convenience).

An important factor was maintainability: as an incoming AI agent (developer), you will find Dramatiq’s codebase and documentation straightforward. The community around it is smaller than Celery’s, but it’s active enough and the feature set covers our needs without excessive bloat. We appreciated that Dramatiq was created with lessons learned from Celery, focusing on being **“pythonic”, efficient, and less error-prone**. For example, Dramatiq’s approach to concurrency (it can use threads or processes) and its message acknowledgement strategy are designed to avoid common pitfalls that Celery users sometimes hit. In summary, Dramatiq offered us a robust yet elegant way to orchestrate background tasks in line with our architecture’s emphasis on reliability and simplicity.

Alternatives considered:

- *Celery:* This is the most well-known Python task queue framework. We gave Celery serious consideration due to its long history and rich plugin ecosystem. Celery certainly could fulfill our needs; it supports RabbitMQ (and Redis, etc.) and has a wide range of features. However, Celery is known to be **complex to configure and can be heavy**. It carries years of legacy design decisions (dating back to older Python versions), and some developers report Celery can be less stable or require more babysitting in production (for example, workers occasionally needing manual restarting, or issues with memory growth). Moreover, Celery’s learning curve is non-trivial; for a new contributor or agent developer, understanding Celery’s nuances (like the need for a separate results backend, how its scheduling works, eventlet vs prefork, etc.) is a bigger ask. Dramatiq, by contrast, markets itself as a simpler alternative that achieves the same core functionality with less configuration. We found in our research and small prototypes that Dramatiq “just worked” out of the box for our use cases, and the code to define tasks was very clean. This simplicity aligns with our guiding principle of minimizing complexity. It’s worth noting that Celery has more out-of-the-box integrations (e.g., with Django, which we’re not using, or monitoring tools like Flower), but those were not decisive advantages. In fact, one source summarized: *“Celery offers extensive features and*

scalability options, while Dramatiq focuses on simplicity and lightweight task execution." – since our scale needs are moderate and simplicity was a priority, we leaned toward Dramatiq.

- *Redis Queue (RQ):* RQ is a simpler task queue that uses Redis as its broker (and storage of results). It's quite straightforward and could have worked for basic asynchronous jobs. We decided against it for a couple of reasons: (1) We had chosen RabbitMQ as the broker for reasons outlined, and RQ doesn't support RabbitMQ – it's tightly bound to Redis. (2) RQ, while easy, lacks some advanced features like built-in scheduling of tasks or robust retry policies (one often has to add additional code or rely on third-party extensions for those). Dramatiq provided those features natively, and since we already were comfortable maintaining RabbitMQ, using RQ would either force us to introduce Redis as a broker (not desired) or not leverage Rabbit at all. So RQ was not an optimal fit here.

- *Huey:* Huey is another lightweight Python task queue (often used with small Flask apps), supporting Redis. Similar to RQ, it's simple but limited and not geared for higher scale or more complex workflows. We dismissed it for similar reasons – lacking RabbitMQ support and fewer features compared to Dramatiq.

- *Custom asyncio loops or cron jobs:* In theory, we could manage background tasks with custom Python asyncio loops or a cron-like scheduler for periodic jobs. This would, however, reinvent a lot of what Dramatiq provides – reliable scheduling, persistence, worker management, error handling, etc. Reinventing that would likely be error-prone. Given the maturity of frameworks like Dramatiq, it made sense to adopt one rather than writing our own task orchestrator.

After weighing these options, **Dramatiq was locked in** as our task execution layer for its combination of reliability, clarity, and integration with our chosen broker. It will allow the Janus agents to perform multiple background operations concurrently and handle long-running tasks gracefully. As you ramp up on the project, you'll find tasks defined as Dramatiq actors throughout the code – these are the units of work that get distributed via RabbitMQ.

Integration note: We set up Dramatiq to use **Redis as a results backend** (via the Dramatiq `Results` middleware). This means when tasks produce a result that needs to be retrieved (e.g., the outcome of an agent's computation), they store it temporarily in Redis for the requester to fetch. This avoids storing result data in RabbitMQ queues themselves. We also enabled Dramatiq's **rate limiting** capabilities for certain tasks. Under the hood, this feature uses Redis and Lua scripting to count task executions and ensure we don't exceed specified rates. For example, if we have a task that calls an external API, we might limit it to N calls per minute; Dramatiq with Redis makes this easy and atomic by using a Lua script to increment and check a counter for the task key. This was an important integration decision: it means Redis not only caches data but also helps coordinate task throttling across distributed workers (ensuring even in parallel, the rate limit is globally respected). We highlight this because it emerged as a requirement during integration – we realized we needed a way to prevent the agent from, say, spamming an API or overloading the system with too many simultaneous heavy tasks. The **Redis + Dramatiq rate limiter** was the solution, leveraging Redis's fast atomic operations (Lua scripts are executed in Redis atomically) to enforce constraints.

In-Memory Cache & Session Store – Redis

Purpose & Role: Redis serves as the **in-memory key–value store** for Janus, handling use cases that require fast, ephemeral storage or pub/sub messaging that doesn't warrant the overhead of the persistent DB or

the formality of RabbitMQ. This includes caching recent computations or frequently used data (to avoid repeated expensive queries), storing session data or short-term state for the agent's interactions, and coordinating transient signals between components. For example, if our agent needs to store the last user prompt temporarily or maintain a short conversation context for quick lookup, it can use Redis. Also, if one part of the system needs to notify another (like a simple event broadcast that doesn't need durability), Redis's publish/subscribe can be used to send a quick message. Essentially, Redis is our **shared, volatile memory** for the architecture, complementing the durable stores.

Why Redis was chosen: Redis is practically the default choice for caching in modern architectures. It's **extremely fast** (in-memory, often completing operations in microseconds) and has a simple but versatile feature set with data structures beyond just plain keys (lists, hashes, sorted sets, etc.). We anticipated various needs like caching LLM prompt results, holding intermediate data for an agent's chain-of-thought, or managing tokens/rate-counters, all of which Redis can handle elegantly. We intentionally separated Redis's role from RabbitMQ's: Rabbit is for guaranteed delivery of tasks, while Redis is for low-latency ephemeral data. This avoids overloading either component with duties unsuited to it (a key design decision was to not use one thing for everything, e.g. resist the temptation to do all queueing and caching in Redis, as that can lead to bottlenecks).

Redis was also chosen because it's easy to deploy and fits our offline model (we can run Redis on the same machine; it's lightweight in terms of binary size and can run with a small memory footprint if needed). It also has persistence options (RDB and AOF files) which means if we want some level of durability for certain caches or session data (in case of process restart), we can enable that. The pub/sub feature of Redis is handy for scenarios where RabbitMQ would be overkill – for example, quickly broadcasting a message to the UI in real-time. Using Redis for such transient pub/sub keeps those messages separate from the durable task queue (which is good for decoupling).

Moreover, many libraries and frameworks (including our chosen ones) offer easy integration with Redis. Dramatiq, as mentioned, uses Redis for its result backend and rate limiting. If we later introduce things like websocket notifications or live dashboards, Redis pub/sub is a common solution to feed those. In short, **Redis was picked for its speed, simplicity, and the way it offloads transient workloads from our other systems**.

Alternatives considered:

- *No dedicated cache (use only the databases or in-process memory):* We contemplated if we truly needed Redis, given we had other stores. Could Postgres or Neo4j suffice for everything? The answer was that using the primary databases for ephemeral data would be inefficient and risk polluting them with data that doesn't need long-term storage. For example, storing short-lived user session info in Postgres would add unnecessary write load and complexity (and slow down retrieval compared to an in-memory lookup). In-process memory (like a Python `lru_cache`) can handle caching in a single process, but in a distributed system with multiple worker processes (and possibly multiple machines), you need a shared cache accessible to all. Redis fills that need perfectly. So a dedicated cache layer was justified for performance and clean architecture.

- *Memcached:* Another popular caching system is Memcached, which is purely an in-memory key-value store (no persistence, no data structures beyond strings). Memcached is very lightweight and could

have been used for simple caching. However, Redis essentially supersedes Memcached in functionality and with equal performance for our purposes. We preferred Redis because:

- It offers more functionality (data structures, pub/sub, scripting), which we anticipated using (and indeed have used in the rate limiter).
- It can persist data if needed (Memcached cannot, so if it restarts you lose all data – that could be problematic for something like session storage).
- We already needed Redis due to Dramatiq's rate limiter and result store, so using it for general caching avoids running a second cache service. In other words, **Redis covered Memcached's use case and more**.
- *RabbitMQ (as a cache or pubsub):* We considered whether RabbitMQ could double up to handle some of the pub/sub or transient messaging. RabbitMQ does support pub/sub style via fanout exchanges, but it's not meant for ultra-low latency or high-frequency messaging where you don't want the overhead of durable queues. Also, using RabbitMQ for ephemeral state (like a distributed cache) is not practical. We kept Rabbit focused on what it's best at (durable queues), and left caching to a system built for speed over durability (Redis).
- *Other KV stores:* There are other specialized stores (like etcd or Consul) but those are more for configuration and service discovery (and are slower in reads compared to Redis). Some newer projects (like Dragonfly) promise to be drop-in Redis replacements with even better performance, but they're not as mature. We stuck with the tried-and-true Redis which is very well supported.

Redis is thus a locked-in component of our stack, dedicated to transient data and quick lookups. One guiding rule we adopted is: *if data can be regenerated or is only needed temporarily, it goes to Redis (not Postgres); if a message doesn't absolutely need guaranteed delivery or needs real-time fan-out, use Redis Pub/Sub (not RabbitMQ).* This clear boundary has helped keep each component doing what it does best.

Object Storage – MinIO

Purpose & Role: MinIO provides the **object storage** layer for Janus. This is where we store large, unstructured binary blobs – files such as documents, PDFs, images generated by the agent, datasets the agent might use, etc. These are pieces of data that are typically larger than what we'd want to keep in a database and do not need to be in memory. Object stores handle things like file streaming, multipart upload, and large binary handling very well. In Janus, if an agent ingests a file from the user or the web, the raw content goes into MinIO (rather than into Postgres or Neo4j) and we keep a reference (like a URL or key) in the database or knowledge graph. Similarly, any output artifacts the agent creates (say it writes a report to a PDF, or creates an audio file) will be saved to MinIO. This separates binary data from our transactional data stores, improving performance and scalability (databases are not good at large blob storage).

Why MinIO was chosen: MinIO is a high-performance, self-hosted object storage solution that is **compatible with the Amazon S3 API**. Essentially, it allows us to run an S3-like service on our own hardware. We chose MinIO because it's extremely easy to deploy (a single small binary) and yet enterprise-grade in terms of capabilities. It supports features like erasure coding (for data redundancy across drives or nodes) and versioning, which can be critical for data durability. Given our off-grid requirement, we wanted

an object store that does not depend on cloud services – MinIO fits that perfectly. It's open-source (Apache License) and has a strong community due to its popularity in on-premise deployments.

The S3 compatibility is a big plus: it means if we use any libraries or tools expecting S3 (for example, an ML model loader or a backup tool), we can point them to MinIO with minimal changes. It future-proofs our architecture – if one day we wanted to migrate to AWS S3 or mix with cloud storage, we can do so because we adhered to a standard API. In the meantime, MinIO gives us full control and runs **entirely locally**. Performance-wise, MinIO is written in Go and is known to be very fast, taking advantage of multi-core systems efficiently for IO. It also has no problem running on ARM, so it can run on a Pi or Jetson, etc., which was necessary for us.

Another factor was that MinIO can scale from a single disk to distributed mode. We can start with a simple single-node deployment, and later, if needed, we could transition to a multi-node MinIO cluster (for example, to get higher availability or aggregate storage from multiple machines). That scalability, combined with the lightweight nature of a single binary, made it an elegant solution. We also considered ease of use: from an incoming developer perspective, dealing with MinIO is straightforward – there's a web UI and the familiar S3 API. All these reasons made **MinIO the locked-in choice for object storage**.

Alternatives considered:

- *Cloud storage (Amazon S3, Google Cloud Storage, etc.):* Using a cloud service was out of line with the offline goal. While integrating directly with AWS S3 would give great scalability, it introduces external dependency and cost, and obviously fails if the system has no internet. We wanted the benefits of S3 *without* relying on AWS – hence MinIO as essentially "self-hosted S3" was the ideal alternative.

- *Ceph (RADOS Gateway):* Ceph is a widely used open-source storage platform that can provide object storage (via its RADOS Gateway which presents an S3 API) in addition to block and file storage. Ceph is highly robust and can scale to petabytes across many nodes. However, Ceph is known to be **complex to deploy and manage**, often requiring a team to tune and maintain it in production. Running Ceph for our scale (initially possibly just tens or hundreds of GBs of data) would be like using a battleship to cross a pond. The overhead in setting up monitors, OSDs (object storage daemons), and so on was unjustified. Ceph shines in large clustered environments with strong multi-node failure tolerance. If Janus evolves into a large distributed system, we might revisit Ceph (particularly if we go Kubernetes, Rook on K3s could manage Ceph for us). But for now, MinIO provides much of the benefit (S3 API, redundancy if needed via erasure coding) with **a fraction of the complexity**.

- *Simple file system or NAS:* Another alternative was to not have a specialized object store at all, and instead use the local file system or a network file share (NFS, Samba, etc.) to store files. This would mean when the agent ingests a file, it just saves it to a directory on disk, and perhaps we index the path in Postgres. While this is very simple, it lacks many features: we'd have to implement our own versioning or deal with name conflicts, handle multi-part uploads, and we wouldn't have an easy HTTP API for retrieving files. Serving files to other processes or potentially to a UI would require setting up a separate server. Essentially, we'd start reimplementing parts of what object stores already do. Also, a single filesystem might not scale well if we later needed more storage than one disk. Considering these drawbacks, we decided a proper object store like MinIO was worth it for reliability and functionality. It gives us an instant REST API to get any object, which is very handy for

integration (for example, an agent can get a file by hitting MinIO's endpoint, or a user interface could retrieve an artifact via a presigned URL, etc.). None of that is as straightforward with just files on disk.

- *Other object storage software:* We briefly looked at other solutions:

- **SeaweedFS:** a distributed file system that also offers an S3 API. It's touted as simpler than Ceph and quite efficient. This was a candidate because it's lightweight and scalable. However, some reports indicated its S3 API compatibility wasn't as full-fledged as MinIO's, and MinIO's single-node performance was better for our expected usage patterns. SeaweedFS excels at huge scale and many small files, but we felt MinIO's focus on pure object storage (and our familiarity with it) gave it the edge.
- **Zenko (by Scality):** an open-source multi-cloud object storage controller that can sit on top of various backends. This was more complex and aimed at hybrid cloud scenarios – not really needed for us.
- We also noted that MinIO has a strong momentum in the industry (with features like adding Kubernetes support, etc.), making it a future-proof choice.

In the end, **MinIO was locked in** for object and blob storage. It ensures Janus can store and retrieve large data pieces efficiently, with the flexibility of S3-compatible tooling. For you as the new engineer, this means if you need to fetch a file that an agent saved, you'll likely be interacting with MinIO (via an API call or an SDK) rather than digging in a database or filesystem manually.

RAG Orchestration and Agents – LangChain + AutoGen + GraphRAG

Purpose & Role: This trio of libraries/tools forms the **Retrieval-Augmented Generation (RAG) and multi-agent orchestration layer** in Janus's architecture. They are not services like the above components, but rather Python frameworks integrated into our code that allow the AI agents to reason, retrieve information, and interact with tools in a structured way. In essence: - **LangChain** provides building blocks to create chains of LLM calls and to easily integrate external tools/data sources into those chains. It's used to structure prompts, manage conversational memory, and call our databases or other utilities from within an LLM-driven flow. - **AutoGen** (from Microsoft) facilitates complex multi-agent dialogues or workflows. It allows multiple AI agents (or agent roles) to converse and collaborate on tasks, orchestrating their interaction. For Janus, this means we can have scenarios like a "Planner" agent and an "Executor" agent working together, or an agent asking another for specific expertise, all within a managed framework. - **GraphRAG** is a technique (and accompanying toolkit) to incorporate a knowledge graph into the RAG pipeline. It complements vector retrieval with graph-based retrieval. Instead of just grabbing documents by similarity, GraphRAG allows the agent to leverage the Neo4j knowledge graph to find relevant context via relationships.

Together, these tools are crucial for bridging our **knowledge stores (Postgres, Neo4j, Qdrant)** with the **LLM reasoning** of the agent. They orchestrate how an agent queries the databases, how it uses the retrieved info in prompts, and how multiple steps of reasoning are chained. The goal is to enable advanced behaviors like iterative problem solving, fact-checking via the knowledge graph, and tool use (e.g., calling a calculator or a web search, if that were allowed).

Why these were chosen: We opted to use existing frameworks like LangChain and AutoGen instead of building everything from scratch because they significantly speed up development and incorporate community best practices.

- **LangChain** has become a de facto standard for creating applications around LLMs. It offers a plethora of integrations: for example, it has modules to interact with SQL databases, vector stores, and even Neo4j. This meant we could hook up Postgres, Qdrant, and Neo4j to the agent with relatively little custom code. LangChain also provides abstractions for agents (LLM-driven decision makers) and tools (functions the agent can call, such as our custom functions to query data). By using LangChain's agent framework, Janus agents can decide when to use a tool – for instance, the agent can be prompted in such a way that it knows it can execute a `search_graph` action that we implement to query Neo4j. LangChain manages the dialogue between the LLM and the tool calls.

- **AutoGen** addresses the need for multi-agent orchestration. While LangChain has some support for multi-agent setups, AutoGen is specifically built to let agents talk to each other in a conversation loop, create sub-tasks, etc. We chose AutoGen to enable scenarios where different specialized agents might handle different aspects of a problem (e.g., a Developer Agent writing code and a Critic Agent reviewing it, a common pattern in self-coding AI workflows). AutoGen handles message-passing, termination conditions of dialogues, and other complexities when you have more than one AI entity interacting. This aligns with Janus's vision of possibly hosting multiple cooperative agents.

- **GraphRAG** was chosen to maximize the use of our knowledge graph (Neo4j) in the retrieval process. Standard RAG might vector-search documents for relevant info, but GraphRAG goes further by pulling in facts from the knowledge graph. We embraced GraphRAG after reviewing research indicating that it can **improve accuracy and explainability** of LLM answers by grounding them in known relationships. For example, if an agent is asked a question that involves multi-hop reasoning (see the example in the GraphRAG paper: connecting a person to their son through historical facts), a pure vector search might not surface those indirect connections, but a graph query can explicitly find the link. GraphRAG allows us to feed the LLM not just a blob of text from vector search, but a structured set of facts/triples from Neo4j as part of the prompt. This hybrid approach (vector + graph) is cutting-edge and we wanted Janus's architecture to support it from the start.

By combining these, we essentially equipped Janus's AI layer with **state-of-the-art tooling for reasoning**. LangChain provides the toolbox and agents, AutoGen manages complex interactions, and GraphRAG injects knowledge graph data into the loop. All are Python libraries, meaning they run within the agent's environment rather than as separate services, but they heavily influence how the agent interacts with our core components.

Alternatives considered:

- *LlamaIndex (GPT Index):* LlamaIndex (now often just called "GPT Index") is an alternative library for connecting LLMs with external data. It allows building various index structures (vector indices, list indices, tree indices, etc.) over your data and can query them to augment prompts. We considered LlamaIndex, as it's quite powerful for document QA scenarios. However, at the time, LlamaIndex was more focused on document retrieval and less on multi-agent orchestration. LangChain's ecosystem was larger and more oriented to agent/tool use cases, which we needed. LlamaIndex might have simplified certain RAG implementations, but we felt LangChain could achieve the same and more,

given our variety of data sources. We did not rule out using LlamaIndex in a limited way (it can be used in conjunction with LangChain), but we decided to standardize on LangChain's abstractions for consistency. A factor was also community and support: LangChain had far more community contributed integrations (many of which we use), whereas LlamaIndex was a bit more niche.

- *Haystack (deepset):* Haystack is an open-source framework for building QA systems. It allows pipelines involving document retrieval (with vector search, etc.), and has components for querying databases or calling APIs. It's more geared towards building a question-answering service, possibly with a fixed pipeline (like retrieve then read). We considered Haystack, but it seemed less flexible for arbitrary agent behaviors and multi-step reasoning loops. Haystack shines if you want a straightforward "give me an answer from these documents" system (with a nice UI and all). For Janus, we required more free-form agent cognition – sometimes the agent might need to plan, use tools in sequence, create new sub-queries, etc. LangChain and AutoGen are better suited to that kind of dynamic flow. Additionally, Haystack didn't (at that time) have integrations with knowledge graphs or multi-agent patterns out-of-the-box, which are central to our design.

- *Custom Orchestration:* We did consider writing our own mini-framework for this: essentially managing prompt templates, tool usage, and multi-agent dialogues with plain Python. While doable, this would have taken significant effort and risk. Reimplementing features like caching prompt results, handling context lengths, or optimizing prompt formatting could divert us from higher-level goals. Given that LangChain and others are open-source, we decided it's better to leverage and perhaps extend these libraries rather than start from scratch. The field of LLM orchestration is moving fast; using a community-supported framework ensures we can pull in improvements (like new prompting techniques or safety checks) as they become available.

- *Other Multi-Agent Systems:* There are a few other projects and research (e.g., Hugging Face Transformers Agents, or proprietary solutions like OpenAI's Function calling with a "planner" agent). We kept an eye on those, but AutoGen had the advantage of being specifically made for complex multi-agent conversations and was open-source. Also, since we were already including Microsoft's research (GraphRAG), using AutoGen (another MS project) made sense as they could complement each other well.

In conclusion, we **locked in LangChain, AutoGen, and GraphRAG** as the core of the AI orchestration logic. This choice means Janus's agents have a powerful capability: they can retrieve information from both vectors and graphs, and coordinate multiple reasoning agents to solve tasks. For an incoming developer like yourself, this means you'll see code structured around these frameworks – with prompt templates, tool definitions, and agent classes provided by LangChain, agent dialogues possibly managed by AutoGen, and specialized query calls leveraging the GraphRAG methodology. We've effectively wired these into the backend components so that an agent can, for example, ask for relevant info from Qdrant and Neo4j in the middle of answering a question, all within a single coherent flow.

Integration Architecture and Component Interplay

Having described the individual components, it's crucial to understand **how they work together as an integrated system**. The strength of the Janus architecture lies in these components complementing each

other to support the AI agent's operation. This section outlines key integration patterns, data flows, and considerations that emerged from combining the chosen stack.

- **Agent Workflow Orchestration:** At the heart of Janus is the AI agent (or agents) which receives tasks (user queries, goals to achieve, etc.) and breaks them down. The **LangChain/AutoGen** logic runs within an agent process, and when the agent decides to perform an action that might be time-consuming or better done asynchronously (like fetching information or doing analysis), it will enqueue a **task via Dramatiq**. For example, if the agent needs to summarize a large document, it might send a task to load and chunk that document. This task is placed onto a **RabbitMQ queue** by the agent, and a **Dramatiq worker** will pick it up, execute it (perhaps interacting with the object store and vector DB in the process), and then return the result (stored in Redis or written to a DB) for the agent to use. This decoupling ensures the agent's chain-of-thought isn't stalled waiting for I/O; it can continue with other reasoning or handle multiple goals by leveraging the queue.

- **Data Retrieval Flow (RAG cycle):** When an agent needs information beyond its prompt context, it will use the **Retrieval-Augmented Generation loop**:

- The agent (via LangChain) formulates a query based on what it needs (this could be a vector query embedding or a graph query or both).
- **Vector search:** If it's looking for relevant documents or text, it will query **Qdrant** through its client API (possibly facilitated by a LangChain `VectorStoreRetriever`). This returns a set of document chunks with similarity scores. The agent then has raw content to work with.
- **Graph search:** If the query involves entities or relationships (e.g., "who is connected to concept X"), the agent will query **Neo4j**. This can happen via a direct Cypher query using Neo4j's Python driver or via a GraphRAG utility that automates the query. For example, the agent might retrieve a subgraph of relevant facts from Neo4j.
- **Combination:** The agent then combines the results – e.g., "documents from Qdrant + fact triples from Neo4j" – and incorporates them into a prompt. This combined context is fed into the LLM (which might be local or an API) to generate the answer or next step.
- **Iteration:** The agent can iterate: maybe the answer from the LLM suggests a follow-up query (LangChain's chain management will loop as needed), or the agent might verify an answer by cross-checking in the graph, etc.

The integration consideration here is ensuring that **identifiers and references** align across systems. For instance, if a document is stored in MinIO and indexed in Qdrant, the Qdrant payload will carry the MinIO object key or a reference ID that the agent can use to fetch the actual document content from MinIO if needed. Similarly, if Neo4j contains a node that corresponds to a document or an entity that also exists in text form, we might store an ID or link so that the agent can move from a graph entity to a document in the object store or a row in Postgres. Maintaining these cross-references (IDs, keys) consistently is an important integration task so that the agent can traverse the **"knowledge triad"** of SQL (structured), Graph, and Vector-backed docs smoothly.

- **Data Ingestion and Indexing:** Another integrated flow is how new data enters the system. Suppose the agent is given a new PDF file to incorporate into its knowledge:
- A background **ingestion task** is created (using Dramatiq) to process the file. This task will save the file to **MinIO** (for long-term storage) and then extract text from it (perhaps using an OCR or PDF parser tool).

- The extracted text is then chunked and **embedded** (here's where the **pending local embedding model** will be used, or currently an API if available). The embeddings and text chunks are inserted into **Qdrant** with appropriate metadata (like "source: PDF X, page 5").
- The task might also extract entities or relationships from the text (if doing GraphRAG indexing). Those facts get inserted as nodes/edges in **Neo4j** to enrich the knowledge graph. For example, if the PDF was about a person, we create a Person node and link them to facts or other entities found in the text.
- Key metadata (like the document title, origin, or a summary) could be stored in **Postgres** (especially if we have a documents table for quick lookup or for listing in a UI).

The integration challenge here is that one ingestion pipeline touches multiple components (MinIO, Qdrant, Neo4j, Postgres). We addressed it by designing a **unified ingestion orchestrator** concept (to be implemented, potentially using a dedicated ETL tool in the future, see pending section on Data Ingestion Orchestrator). In the interim, our Dramatiq tasks serve as the orchestrator: a single task can call all needed services in sequence. We ensure **idempotency** where possible (so if a task fails halfway, rerunning it won't duplicate data in Neo4j or Qdrant – using unique IDs or checks).

- **Consistency and Sync:** Because we have some overlapping data across systems, part of integration is defining which system is the authority for a given piece of data. For example, if we have a user profile, it likely lives solely in Postgres (with maybe a user node in Neo4j only if needed for graph relationships). If we have a piece of knowledge, it might exist in both Neo4j (structured facts) and Qdrant (as an embedding of a description). We must ensure that if that knowledge is updated or removed, we update/remove it in both places. This is currently managed by the application logic – e.g., an agent or admin action that deletes a knowledge item would call both databases. We flagged this as a place where a **unified data access layer or federation** would help (to avoid mistakes where one store is updated and another is not). In the current phase, careful documentation and encapsulation of data operations mitigate issues (for example, all code that adds knowledge goes through a function that writes to both Neo4j and Qdrant). In the next phase, implementing the Unified Access API will formalize this and reduce potential inconsistency.

- **Backpressure and Throttling:** As touched on, integration of RabbitMQ and Dramatiq with Redis allows us to handle backpressure. For instance, if the agent generates tasks faster than workers can handle (imagine it decides to scrape 100 web pages at once), RabbitMQ will queue them but we don't want unlimited growth. We set up policies (like a maximum queue length or TTL on certain queues). Dramatiq's rate limiter with Redis ensures that certain classes of tasks won't be executed more than N at a time. We use Redis counters (with Lua for atomicity) to implement a **global concurrency limit** on some tasks – e.g., "only 2 web-scraping tasks may run simultaneously to conserve bandwidth." This interplay between RabbitMQ (queuing), Dramatiq (workers), and Redis (rate limit state) was a deliberate integration solution to avoid overload and to comply with any external rate limits. It highlights how no component alone solved the requirement, but together they did.

- **Observability Hooks:** In integrating everything, we also added hooks for future observability. For example, RabbitMQ can export metrics (queue lengths, consumer counts), and we plan to collect those. Dramatiq can emit logs or events for task successes/failures. Postgres logs slow queries. Neo4j has query logging and metrics. We aren't fully aggregating these yet (pending observability stack), but the architecture is prepared in that each component's standard monitoring endpoints will be utilized. We mention this because integration isn't just about data flow, but also about how to **manage the system as a whole**. We ensure each piece exposes the info needed to supervise it so

that the orchestrator agent or devops can get a holistic view (e.g., if an agent is running slow, we could check and find RabbitMQ is backed up, then trace which tasks are jamming it, etc.).

- **Security and Access Control Integration:** Currently, all components trust each other on a secure network (or localhost). We integrated minimal authentication (each service typically has a user/pass set – e.g., Postgres credentials, RabbitMQ user, Neo4j user, MinIO access key/secret). These are stored in configuration and used by the agent services. Looking forward, we identified that as we integrate a potential user-facing API or UI, we need a unified authentication layer – likely the **IAM component** (pending). For now, integration is simpler: the AI agent code has high-level access to everything via stored creds. But we've containerized services and set network rules such that, for example, the MinIO admin console isn't exposed publicly, and the database isn't accessible from outside. This is part of integration in terms of deployment: making sure that adding all these parts doesn't inadvertently create security holes. We will strengthen this with Vault or IAM integration in the future, but the groundwork is in place (all internal connections use credentials and, where possible, TLS within the environment).

- **Deployment and Environment:** Integration also required thinking about how to run all these services together. We use Docker Compose in development to spin up RabbitMQ, Redis, Qdrant, Neo4j, MinIO, etc., and connect them on a common network. We've fixed environment variables and configuration such that services know how to reach each other (e.g., environment for Dramatiq worker contains the RabbitMQ URL and Redis URL, etc.). Ensuring version compatibility was another integration chore – for example, making sure our Neo4j Python driver version matches the Neo4j database version for smooth Bolt protocol operation, or that the Qdrant client and server versions align on API. We've documented these and pinned versions where needed. The idea is that a new agent (like yourself) can start the whole stack with one command, and all components will come up and interconnect. This also aids testing: we can run end-to-end integration tests by bringing up the stack and simulating an agent workflow, then inspect that data landed in the right places.

In summary, the interplay of components in Janus is orchestrated so that each does what it is best at: **PostgreSQL guards the core data**, **Neo4j stores and exposes relationships**, **Qdrant provides semantic search**, **RabbitMQ + Dramatiq ensure tasks are executed reliably**, **Redis speeds up transient interactions**, **MinIO holds bulk data**, and **LangChain/AutoGen/GraphRAG ties it all together for the AI logic**. The architecture avoids any single point of overload by distributing responsibilities. For instance, large files don't clog the database (they go to MinIO), frequent short requests don't hammer Postgres (they might be cached in Redis), heavy computations are offloaded from the main agent loop (via RabbitMQ tasks), and multi-hop reasoning is offloaded from the LLM to the knowledge graph (via Neo4j queries).

One concrete example to illustrate integration: *Suppose the user asks the agent: "Find the connection between quantum physics research and renewable energy innovations, and write a summary."* The agent will: 1. Use LangChain to interpret the request and plan steps. 2. It might first query Neo4j: find if our knowledge graph has any direct links between "Quantum Physics" and "Renewable Energy" topics (perhaps a Neo4j query result shows a connecting entity, like a person or organization working in both fields). 3. It also formulates a vector query to Qdrant to find relevant documents (maybe articles or papers in the object store) on those topics. 4. Those documents are fetched from MinIO (the agent retrieves them using the keys from Qdrant results). 5. The agent may spawn a **Dramatiq task** to summarize each long document so it doesn't have to feed the entire text into the LLM (this task reads from MinIO, uses an LLM summarization routine, stores the summary in Redis or returns it). 6. Meanwhile, the agent collects facts from Neo4j (GraphRAG might supply

that, e.g., “Person X is a physicist who applied quantum algorithms to improve solar panel efficiency in Project Y” from the graph). 7. Once document summaries are back (the agent either awaited them or checks results in Redis), the agent composes the final summary by prompting the LLM with both the **graph facts and the document summaries** as context. 8. The answer is given to the user. The agent then might **store the newly synthesized knowledge**: perhaps it adds a node in Neo4j that links Quantum Physics and Renewable Energy through the summary (or stores the summary text in Postgres for record-keeping). 9. Throughout, any errors (say a failed task or missing data) are logged; if RabbitMQ had a backlog, the system could scale by adding another worker process or the agent would see a delay and could even inform the user it’s working.

This scenario shows multiple components in action for one user query, demonstrating the integrated nature of the system. The design ensures the agent can leverage structured, unstructured, and vector knowledge together, which is the core promise of Janus’s architecture – to be an orchestration platform where an AI can **see through multiple “eyes” (SQL tables, graphs, vectors) and act through multiple “hands” (background tasks, tools)** to achieve its goals.

Pending Components and Open Questions (Next Phase)

While we have solidified the core backend stack, a number of **components remain pending or questions remain open**. These are slated for deeper research and decisions in the next phase of the Janus project. It’s critical to address many of these to fulfill the platform’s vision and to ensure our architecture remains coherent as we add more capabilities. Below we enumerate these pending items, why they matter, and what needs to be finalized:

- **Local Embedding Generator (Vectorization Model):** *How will we generate embeddings on-premise?* Currently, our vector database Qdrant is most useful when we can populate it with embeddings from data. Up to now, we may have been relying on an API (e.g., OpenAI’s embedding service) or a placeholder method for embedding. To truly go off-grid, we need to choose and integrate a **local embedding model**. This could be a HuggingFace Transformer model (like `all-MiniLM-L6-v2` or similar) run via PyTorch, or perhaps a more optimized solution like using the SentenceTransformers library with a quantized model or using a smaller LLM to get embeddings. The key requirements are that it runs reasonably fast on available hardware (possibly without GPU, though if GPU is present we’ll utilize it) and produces embeddings of sufficient quality for our tasks (semantic search, etc.). We’re exploring models in the 384 to 768 dimensional range that are known to work well in general domains. Another angle is using a model that can run on CPU efficiently (e.g., using INT8 quantization or a smaller architecture like DistilBERT). The decision here affects how we integrate with Qdrant: we will need to pipeline data through this model during ingestion and also embed user queries on the fly for similarity search. The research will involve testing a few models’ accuracy vs performance trade-offs. Until this is finalized, Janus might have a dependency on external embedding services, which is not ideal for the fully autonomous goal. So this is a **top priority to finalize**. Once we select a model, we’ll containerize it or incorporate it into the agent runtime and ensure that the embedding generation is seamless (likely adding a service or an asynchronous task for heavy batch embeddings).

- **Local LLM Engine (Off-grid LLM):** *Which large language model will Janus run locally to power the agents’ reasoning?* Right now, if Janus uses an API like GPT-4 for its intelligence, it violates the off-grid

premise and adds cost/dependency. We plan to integrate a local LLM. Options include running an optimized model via **llama.cpp** (for Llama family models) or using **vLLM** or DeepSpeed to serve a larger model on GPU if one is available. There are also orchestrators like **Ollama** that simplify hosting models. The choice might be a smaller model (7B to 13B parameters) that can run on CPU with quantization (e.g., 4-bit quantized Llama2 13B might fit in 16GB RAM). We need to assess what model provides acceptable capability for our use – maybe Llama-2, GPT-J, or newer open models. The guiding principle is that the model must run **fully offline** on at most the kind of hardware we anticipate (a powerful single PC or a small server). This might require some compromises on quality vs the top-tier online models, but it ensures autonomy. Once chosen, integration steps will involve connecting this LLM to our LangChain/AutoGen setup (LangChain supports local HuggingFace models or llama.cpp models via wrappers). We should also set up a mechanism for model updates or swaps (so if a better model comes out, we can upgrade). An open question is the interface: do we run the model as a local API server that the agent calls (to decouple the process), or load it in-process? In-process might yield lower latency but can consume a lot of memory in the agent process. This is a design decision we'll have to make. **Finalizing the local LLM** is critical for the next phase because it unlocks true offline functionality and removes reliance on external AI. It's also one of the riskier pieces (in terms of whether the chosen model will be "good enough"), so it may require experimentation and possibly fine-tuning on our domain, which we have planned for.

- **Document Store:** *Where do we keep unstructured or semi-structured data that isn't in the graph or object store?* As we ingest data, not everything is a binary blob or a neatly structured record. There might be JSON data, scraped web pages, intermediate representations, or simply a need to quickly dump data somewhere. A **document store** refers to a database optimized for storing documents/JSON with flexible schema – essentially a NoSQL store (could be MongoDB, CouchDB, or even something like SQLite with LiteFS for distribution). We haven't finalized this slot because we were initially focusing on core needs. However, as Janus grows, having a document store could be very useful. For instance, if the agent crawls a webpage, we might store the raw HTML or parsed text in a doc store, from which we then create vector embeddings and knowledge graph entries. The doc store would serve as a **staging area or archive** of raw data. We listed candidates:

  - **CouchDB:** which is appealing for its offline-first replication (Couch can sync between nodes or to mobile devices, etc., which is interesting if Janus instances collaborate).
  - **MongoDB Community Edition:** very popular and flexible, but its license (SSPL) is something to note – however, for our internal use it's fine. Mongo has a rich query language for JSON which could be handy.
  - **LiteFS with SQLite:** a lighter approach where we use an SQLite database for documents and replicate it using LiteFS (by Fly.io) for distribution. This is a bit experimental but could keep things very lightweight.

The decision will hinge on how much unstructured data we expect and the need for replication. At this moment, we're using Postgres and the object store to fill the gap (e.g., storing some JSON in Postgres or storing text files in MinIO). That's workable but not optimal. So, to improve our ingestion pipeline and possibly to support things like fast full-text search on raw text (unless we use an external search engine), we should decide on a document store. This isn't as urgent as embeddings or LLM because we have workarounds, but it is important for **archival continuity** – ensuring that as the agent ingests more data,

we're not losing or mishandling the original sources. Mark this as a key item to address in the next phase once the AI basics are in place.

- **Time-Series Database:** *Do we need a specialized time-series solution for metrics or sensor data?* We identified a slot for a time-series DB because Janus could be used in contexts involving IoT or physiological data (the vision mentions "physiological sensor data"). If the agent will ingest streams of timestamped data (e.g., CPU metrics, or readings from a sensor network), storing that efficiently is something time-series databases excel at (with compression, downsampling, fast range queries). **TimescaleDB** (a Postgres extension) was our leaning because it would integrate nicely with PostgreSQL (and could even be used within our existing Postgres instance to avoid a new service). Alternatives like **InfluxDB** or **VictoriaMetrics** are standalone time-series DBs. This component remains unfilled because we need to clarify the requirements: if Janus's focus is more on document and knowledge processing, we might not introduce this yet. However, if in the next phase we plan to monitor a bunch of system metrics or have the agent analyze trends over time (like logs, user interactions over time, etc.), a time-series DB would be very useful. The decision point will be: are we starting to accumulate data that is naturally time-indexed and benefits from specialized storage? If yes, Timescale (as an extension) might be the easiest path (just requires running the Timescale version of Postgres or installing the extension), which keeps our component count the same. This is not critical-path for core functionality, but it is for completeness of the "resilience and monitoring" goal – e.g., an observability stack might use a time-series DB to store metrics anyway.

- **Event Streaming / Event Store:** *How will we record or publish events in an immutable log?* We have RabbitMQ for messaging, but there's a concept of an **Event Store** (as used in CQRS/Event Sourcing patterns) – a log of all events (state changes, actions taken by agents, etc.) that is never altered. This can be useful for auditing, debugging, or replaying sequences. We've noted options like **EventStoreDB** (a specialized event store database), or using Postgres (with a table that appends events), or even leveraging **Redis Streams** for a simpler event log. This component is pending because it depends on whether we adopt an event-sourced approach internally. As of now, we do log actions (e.g., we might log agent actions to Postgres), but we haven't formalized it as a separate store. In the next phase, if we find that debugging agent behaviors or providing a history of decisions is important, implementing an event store could be very helpful. For example, every time the agent makes a significant decision or every time a user interacts, we append an event. This chronological event log can feed into learning (the agent could analyze its past events) or recovery (if the system crashes, events can rebuild state). The open question is how far we want to go with event sourcing. A lightweight path is to use the **Postgres commit log or a dedicated table**, which is simplest. A heavier but more scalable path is something like EventStoreDB or even Kafka (with topics representing event streams for different entity types). Given our offline constraint, EventStoreDB (which can run self-hosted) or Postgres are likely candidates over Kafka. This decision will be made once we scope out how critical auditability and replayability of events is in the near term. It's marked as an open possibility that can significantly improve **traceability** of agent actions.

- **Unified Access API:** *How do we simplify querying across multiple stores for consumers (like UIs or external programs)?* As mentioned, currently if one needed to gather information that spans SQL, graph, and vector, one would have to query each separately and merge results in code. A unified API would abstract that. We listed possibilities like **Hasura** (GraphQL on Postgres) extended with custom resolvers for Neo4j/Qdrant, or a **GraphQL federation** approach that combines multiple GraphQL endpoints, or a custom **FastAPI/GRPC service** that provides custom endpoints (like an endpoint `/`

`knowledge_search?q=` that internally hits all relevant sources). This is pending likely until we build a user-facing dashboard or developer API. However, it's critical to start designing it now so that our data models align. For example, if a document has an ID in Postgres and the same ID is used as a key in Qdrant and as a node property in Neo4j, then creating a unified API is easier because you can join on that ID across sources. We have been conscious of that in the current design (maintaining cross-references). In the next phase, as soon as an external need arises (like a web UI that wants to query what the agent knows about X), implementing a unified API should become a priority. The open questions revolve around technology choice: GraphQL is attractive for its single-query that can fetch from multiple sources if set up correctly, but it might add complexity. A simpler approach might be a small set of REST endpoints that perform common composite queries. For instance, an endpoint `/agent/memory?topic=X` might under the hood query Neo4j for relationships on X, query Qdrant for relevant docs on X, and return a combined JSON. We will likely prototype a bit and choose an approach that is both developer-friendly and matches our team's expertise. Regardless, this component will be important for **agent interoperability** (if another system or agent wants to access Janus's knowledge easily) and for eventual UI integration.

- **Observability Suite (Logging/Monitoring):** *How do we gain insight into the running system's health and performance?* This is a must-have for moving from prototype to production-ready. We plan to integrate a combination of:

- **Centralized Logging:** using something like **Loki** (Grafana Labs' lightweight log aggregation) or the ELK stack (Elasticsearch for logs). Logging integration means all components (our agent app, the databases, etc.) should send their logs to a central place. We have to ensure logs are parseable and tagged with source. For example, run Redis and RabbitMQ with logging to stdout and have our log collector ingest those.
- **Metrics & Alerts:** using **Prometheus** to scrape metrics. Many of our components natively expose metrics endpoints (RabbitMQ's management plugin, Neo4j has JMX metrics or can be extended, even Qdrant and MinIO have endpoints). Prometheus can gather these, and **Grafana** can visualize them. We will set up dashboards to monitor queue lengths, memory usage, query throughput, etc. We'll also create some custom application metrics (like number of tasks processed, rate of agent queries, etc.).
- **Distributed Tracing:** if applicable, using OpenTelemetry to trace request flows across components. This might be more advanced and may require instrumenting our code and possibly using an OpenTelemetry Collector and a backend like Jaeger. This is nice-to-have for debugging complex interactions (e.g., following a user query through the agent's steps and see timing).

In the next phase, implementing the core of this (logging and basic metrics) is critical to maintain our **resilience principle** – you can't have resilience without visibility. During integration, we identified where we can hook in: e.g., Dramatiq can log task events (failures, durations), RabbitMQ gives queue stats, etc., so we know it's feasible. The decision we need to finalize is which stack to use. **Grafana/Loki/Prometheus** is appealing because it's open-source and can be run offline, and it's not too heavy for a single machine. We might also look at **OpenTelemetry** as a unified approach for metrics and logs. But likely a combination of Prom+Grafana (for metrics) and Loki (for logs) is straightforward. We'll containerize those and include them in the deployment. Alerts can be configured (e.g., if CPU is high or queue is growing, etc.) – perhaps using

Grafana's alerting or Prometheus Alertmanager. This isn't an "end-user" feature, but as the agent caretaker, you (and future developers) will rely on it heavily to understand system behavior and catch issues early.

- **Configuration & Secrets Management:** *How do we manage configuration (especially secrets) across environments?* Right now, our configuration (like database URLs, API keys, etc.) might be sitting in `.env` files or docker-compose files. That's fine for development but doesn't scale to multi-node or higher security needs. We plan to integrate a tool like **HashiCorp Vault** for secrets, or a simpler approach if this remains single-node (maybe environment-variable management with something like Doppler or even an encrypted config file using SOPS). The key open question is deployment scope: if Janus is deployed on multiple machines, a central secrets manager (Vault) is justified – it can distribute secrets and even manage dynamic credentials (e.g., rotate the Postgres password). If on a single node, Vault might be overkill and a simpler solution (like docker secrets, or just well-secured env files) could suffice. We lean towards at least using **Vault in dev** to practice, because it's a robust solution and aligns with resilience (it can store secrets encrypted and control access). We'll need to adjust our applications to fetch secrets from Vault at startup (or use Vault agent injection, etc.). This is a task to finalize soon because it touches on how we deploy everything securely. We want to avoid secrets in plain text in git or images, obviously. The outcome might be that in the next phase, we run a Vault server (or even a lightweight variant) alongside Janus, or we integrate with a cloud secret store when bursting to managed services.

- **Backup & Disaster Recovery:** *How do we backup the data stored in Janus's multiple components?* We must plan for worst-case scenarios like hardware failure or data corruption. Each component requires a backup strategy:

- **PostgreSQL:** We should schedule regular dumps or use continuous archiving (WAL archiving) to be able to restore the database to a recent point. Tools like `pg_dump`, `pg_basebackup` or even something like **Barman** can be used. Since we want automation, we might script a nightly backup (and possibly store the backups in MinIO or an external drive).
- **Neo4j:** For the community edition, backups can be done by copying the database files when the DB is shut down, or using the `neo4j-admin backup` tool if available (that might be enterprise only). We might instead use Neo4j's APOC triggers to periodically export portions of the graph to files. This needs research. At minimum, we could stop Neo4j daily (in a maintenance window) to snapshot it. Not elegant, so we'll see.
- **Qdrant:** We can use Qdrant's snapshot capability (it has API to create snapshots of the vector index). These snapshots could be stored in MinIO or offsite. Qdrant is relatively small data (embeddings) compared to raw docs, but still important.
- **MinIO:** Since MinIO holds potentially irreplaceable files, we should back it up as well. If running single-node, enabling versioning on buckets is one safeguard (accidental deletions can be recovered). But for true backup, we could use a tool like **Restic** or **Rclone** to copy MinIO buckets to an external storage (like an attached HDD or cloud bucket if allowed). Another approach is to run MinIO in erasure-coded mode across multiple disks so it's resilient to disk failure (but that's within the system). We might do both: local redundancy and external backup.
- **Redis:** Typically we don't backup Redis data because it's ephemeral (though if we use it for anything critical, we might enable AOF persistence so it can recover state on restart). But backup per se is not needed for caches.
- **RabbitMQ:** We also usually don't backup message queues – if the system fails, any queued tasks can be recreated or handled manually. We assume an acceptable loss of queued but not-yet-processed

tasks if the whole system goes down, as long as the underlying data is safe. However, RabbitMQ's definitions (like queue configurations or user permissions) could be exported as needed. In practice, we can reconstruct those easily or treat them as code (in config).

We flagged backup as essential because of our resilience goal: an off-grid system might be running in a harsh environment (imagine a field lab) where hardware could fail. We want Janus to be able to recover or at least not lose all knowledge. The next phase should see implementing a **backup schedule** (perhaps via cron jobs in the host or a Kubernetes CronJob if we go that route). We mentioned tools like **Restic, BorgBackup** – these can deduplicate and encrypt backups, which is great if we plan to ship backups off-site (maybe to a secure cloud location or an external drive). **Velero** was mentioned for Kubernetes backups, but that's if we containerize under k8s. For now, a simpler approach with restic might do. We'll finalize which tool fits best. Possibly, we'll store backups in an encrypted form on an external medium, to be rotated. The open question is how automated we want it and where backups reside (purely offline local backups vs occasionally connecting to upload to a remote safe location – a policy decision).

Also, **disaster recovery procedures** should be documented (e.g., how to restore a Neo4j from backup, how to re-init Qdrant with snapshot, etc.). Finalizing backup is crucial before we accumulate too much unique data; ideally in the next phase we implement at least a basic backup for each core component.

- **Edge Deployment / Multi-Platform Support:** *How will we deploy Janus uniformly across different hardware setups?* Currently, in development we use Docker Compose. For production or varied environments, we consider:
- **Docker Compose Profiles:** This can allow toggling certain components on/off (like maybe not running some pending ones if not needed). It's simple and good for single-node.
- **K3s (Lightweight Kubernetes):** If we foresee running Janus on something like a cluster of Raspberry Pis or mixing devices, K3s could give a consistent environment. K3s together with something like **Rook-Ceph** could even provide a distributed file system for MinIO or other uses.
- **NixOS Flakes / Containers:** A declarative approach (Nix) could ensure all dependencies and config are consistent across environments. It's a bit niche but powerful for reproducibility.

The main integration point here is making sure our solution is **hardware-agnostic**. All chosen software has ARM builds (we explicitly ensured that for resilience). We tested things on a Raspberry Pi for example, to confirm. Edge deployment might involve specialized hardware toggles, e.g., enabling GPU use when available (we'll design config such that if a GPU is present, the local LLM or embedding model can use it, otherwise run CPU). Docker helps abstract away OS differences to an extent (multi-arch images).

The open question is whether to stick with Compose or adopt Kubernetes in the next phase for more complex deployments. If our next phase goal is to demonstrate Janus on multiple devices or easily shift from local to cloud, container orchestration might be worth it. If we remain on one node, Compose is fine. We've left this open with the note of possibly using **K3s + Rook-Ceph** for a uniform experience across a cluster (Ceph could handle storage for MinIO and possibly Postgres, but again at cost of complexity).

For immediate next steps, a pragmatic route might be: refine the Docker Compose setup (maybe break it into dev and prod versions), and if time permits, create a Helm chart or k8s manifests for those who want to deploy on Kubernetes. This would finalize the *deployment architecture* which goes hand in hand with the software architecture.

- **Miscellaneous Open Questions:**

- *Neo4j Scalability:* As noted, if our graph grows huge or we need multi-user high availability, we either consider Neo4j Enterprise (which is a licensing cost issue) or an alternative like Neptune or a different graph DB. This is not pressing for the next phase unless our use case pivots to a multi-client server scenario. But it's worth monitoring. Our mitigation for now is that we don't push Neo4j beyond Community limits (single instance with up to maybe a few million nodes/relationships, which is typically fine).
- *AutoGen and GraphRAG Stability:* These being relatively new, we need to keep an eye on their updates. An open question is how to continuously integrate improvements or switch out if needed (e.g., if LangChain introduces multi-agent features that could replace AutoGen, or if GraphRAG gets an official library/toolset we should adopt).
- *Experiment Tracking:* If Janus's development leads to training models or doing experiments (like fine-tuning the local LLM), we would want to integrate an **Experiment Tracker/Model Registry** (e.g., MLflow or Weights & Biases self-hosted). We flagged this in the pending list. It's not critical for the agent's functioning, but for the developers to keep track of model versions, datasets, and results, it's very useful. Next phase, if we embark on fine-tuning, we should deploy MLflow (which could run alongside and even use the MinIO or Postgres for storage).
- *Plugin/Tool Ecosystem:* As the agent grows, we might integrate more "tools" (e.g., the ability to call external APIs or run code). We should maintain a modular way to add these. LangChain helps with that (tools are just classes). No immediate question, but noting that as integration of *external* tools (like maybe a web-browsing tool or email-sending tool) would introduce new components (like needing an email server or internet access) – those we would consider case-by-case with the off-grid constraint (e.g., maybe an SMS gateway via a connected device).

Finally, a **recap of priorities for next phase** (which can serve as a guide for you, the incoming agent): 1. **Finalize Embedding & LLM choices** – get those running locally, as they unblock many other things. 2. **Implement Observability and Backup** – protect and monitor what we have before expanding further. 3. **Address open slots by criticality**: Document store if ingestion is ramping up, Unified API if a UI or external usage is planned soon, IAM if multi-user access is on the horizon. 4. **Test resilience** – do drills like unplug the network (simulate off-grid), crash components to see recovery, restore from backups, etc., to validate our guiding principles in practice.

Each pending item carries its rationale from above and ensures continuity of the design philosophy: **maintaining autonomy, reliability, and extensibility** of the Janus backend. By tackling these in the next phase, we will move from a well-architected prototype to a robust platform capable of supporting advanced AI agent operations in diverse environments.
