ChatGPT

Backup and DR Strategy for the Janus Backend

Requirements and Constraints

Janus's backup solution must meet several strict requirements, given the current single-node setup and future plans to scale to a cluster. Key needs include:

- **Multi-Data-Source Support:** Automated **scheduled backups** for all backend components – PostgreSQL, Neo4j (Community Edition), Qdrant, MinIO object storage, Redis (with AOF persistence), and RabbitMQ (config files). Each service may require a tailored backup method (e.g. SQL dumps for Postgres, snapshot APIs for Qdrant, file copies for config).
- **Offline-First & Secure:** The environment is largely offline. Backups must be **encrypted** and stored locally, with optional synchronization to external media or an on-prem object store (e.g. a MinIO bucket). The system should tolerate limited connectivity and ensure data confidentiality (encryption at rest by default).
- **Self-Hostable & Lightweight:** The solution should be fully self-hosted with minimal external dependencies or cloud services. It should run on bare-metal or in Docker containers, **not requiring Kubernetes** in this phase. Simplicity and reliability are valued over complex infrastructure.
- **Automated Verification:** It's not enough to take backups – we need confidence they work. The tool/stack should enable **automated restore testing or backup integrity verification** on a schedule. This might include checksum verifications, periodic test restores, or spin-up of services from backups to verify data integrity.
- **Low Maintenance & Declarative Config:** Aim for a "set-and-forget" system with **minimal manual intervention**. Backup jobs for each service should be defined in a **declarative** manner (as config files or code) rather than ad-hoc scripts, making it easy to track and adjust schedules or retention. Monitoring and alerting on failures is important for visibility.
- **Container Compatibility:** The backup process must work in a containerized environment (Docker), since Janus components run in containers. It should interact safely with running containers (e.g. via exec or temporary pauses) to get consistent snapshots. However, **no Kubernetes is required yet** – we want to avoid the complexity of a full K8s backup operator at this stage.
- **Future-Proofing for K8s:** In later phases, Janus will migrate to Kubernetes. The chosen approach should offer a clear migration path – for example, by aligning with tools like Velero (the K8s backup solution) or by using methods (CronJobs, CSI storage snapshots) that can be translated into a K8s context. The goal is to avoid a dead-end solution now that cannot transition to cluster backups.

Candidate Backup Solutions Overview

We researched a broad range of backup tools and stacks – extending beyond well-known options like Restic, BorgBackup, or Velero – to find the best fit for Janus. The solutions considered fall into a few categories:

- **Deduplicating Backup Tools (CLI-based):** Modern file-level backup utilities such as **Restic**, **BorgBackup**, **Kopia**, and **Duplicacy**. These focus on efficient, encrypted backups with deduplication,

but typically require external scheduling/orchestration. Wrappers like **Borgmatic** or **Backrest** add scheduling and ease-of-use on top of these engines.

- **Backup Orchestrators / Suites:** Integrated systems that coordinate backups across multiple data sources. For example, **SHIELD** (open-source backup orchestrator) and legacy enterprise tools like **Bacula/Bareos** or **Amanda**. These often include scheduling, retention policies, and plugins for databases. They are powerful but can be complex to set up.
- **Container-Native Backup Tools:** Utilities designed for Docker environments, such as **Offen's docker-volume-backup** and **Volumerize**, which run as sidecar containers to snapshot volumes. These emphasize ease of deployment in containers (and often handle container stop/start for consistency). They may trade off advanced features (like deduplication) for simplicity.
- **Kubernetes-Focused Tools (for future):** Since K8s is on the horizon, we note tools like **Velero** (with restic or CSI plugins) and **Stash** (K8s operator for backups). These are not usable in the current non-K8s setup, but inform how our chosen path can transition later.

Below, we compare several promising candidates on how they satisfy Janus's requirements:

Comparison of Backup Tools/Stacks

Solution	Backup Mechanism & Scope	Automation & Scheduling	Encryption & Storage	Restore Verification	Overhead & Maintenance	K8 Pa
BorgBackup + Borgmatic	File-level incremental backups with global deduplication (Borg). Borgmatic wrapper handles multiple sources (files, DB dumps). Suitable for all services by dumping data to files and backing up directories [1].	Declarative YAML config; Borgmatic schedules via cron or systemd. Supports per-job hooks (e.g. run pg_dump before backup) [2]. Retention policies are configurable and backups can run unattended.	**Encryption:** Borg supports client-side AES-256 encryption. **Storage:** Repos on local disk or SSH. (No native S3; would require mounting or syncing to MinIO). Offsite sync can be done via `rclone` or external disk [3] [4].	Borgmatic can run `borg check` to verify repository and archives. No built-in full restore test, but supports mounting backups for spot-checking. Automated test restores would need scripting.	Lightweight (Python + Borg). No separate server required. Low maintenance once configured; jobs defined in config. Monitoring via Healthchecks or notifiers (integrations for uptime checks and alerts on failure) [5] [6].	Bo K8 to cl (K di tr th co re lo st re re Ve re

Solution	Backup Mechanism & Scope	Automation & Scheduling	Encryption & Storage	Restore Verification	Overhead & Maintenance	K8 Pa
Restic + Backrest	File-level incremental backups with deduplication (Restic). Backrest adds a web UI and acts as a scheduler/ orchestrator for restic [7]. Can backup Docker volumes and host paths; supports hooking container stop/start for consistency [8]. All Janus data can be included by mounting volumes or using pre-backup scripts (e.g. call Qdrant snapshot API, dump Neo4j offline).	Backrest provides an internal scheduler (cron-like) via its always-running service [7]. Jobs (backup "plans") are configured in the UI or JSON config, including schedules and retention. No external cron needed – Backrest runs as a lightweight daemon [9] [10].	**Encryption:** Restic encrypts backups by default (AES-256 & Poly1305 HMAC) [11]. **Storage:** Restic supports local folders or S3-compatible object stores natively [12]. (Backrest can easily target a MinIO bucket for offsite sync).	Restic has built-in `restic check` for repository integrity. Backrest UI makes it easy to trigger restores and even browse files in snapshots [10]. However, fully automated restore testing would still require custom scripting (e.g. periodic restore of a snapshot to a test environment).	Moderate overhead: Backrest is a single Go binary (~20 MB) running in a container or as a service [9]. It introduces a UI and config files, but is self-contained. Web UI simplifies operations (good observability via dashboard). Newer project (as of 2024) – less proven than Borgmatic. Minimal manual intervention after setup (scheduling and pruning handled internally).	Re w ba **Ve** in re re PV fu de co co th re [1] "r in w se pe co si sh re ca ou its w us (V re fu

Solution	Backup Mechanism & Scope	Automation & Scheduling	Encryption & Storage	Restore Verification	Overhead & Maintenance	K8 Pa
Kopia (CLI)	File-level incremental backups with deduplication and compression (Kopia). Can back up directories or whole volumes. Would require custom scripting to dump databases and invoke Kopia backup commands. Good for capturing all data types (files, dumps) similarly to restic.	No native scheduler daemon. Use OS cron, systemd timers, or a simple script to run Kopia on schedule. (Kopia has a GUI and a "server" mode for central management, but for automation we'd script it). Policies can enforce retention within the repository.	**Encryption:** Always enabled – Kopia encrypts all data client-side (password or key-based) [13]. **Storage:** Supports local disk, S3/ MinIO, B2, etc. (similar flexibility to restic) [14].	Kopia performs automatic repository health checks and cleanup of expired snapshots [15]. It can verify backup integrity. No out-of-the-box restore testing; would need a periodic manual restore procedure or scripted validation of data hashes.	Lightweight binary (Go-based). Running Kopia requires managing config and credentials but no persistent service. It's relatively new but noted for high performance – tests show **up to 7x faster** backups and **20% less space** vs. restic [16]. Maintenance is moderate: upgrades are straightforward, but no GUI unless you run Kopia's optional server.	Ko **su Ve** ar ba to Th tr co Ko ba ac no la w Ko in (g co ba Th qu ho Ve su m

SHIELD (Backup Orchestrator)	**Unified backup system with plugins** for various data systems. Shield can schedule and run backup jobs for Postgres, MySQL, MongoDB, Redis, etc., using the proper native tools (e.g. it uses `pg_dump` for PostgreSQL) [18]. It stores backups in archives (tarballs) and can handle restores. Lacks a built-in Neo4j or Qdrant plugin, but those could be backed up via the generic filesystem plugin (e.g. run a pre-script to dump Neo4j, or copy Qdrant snapshot files). RabbitMQ config can be grabbed as	Shield includes its own scheduler and retention policies via a web UI or CLI. You define “targets” (e.g. a Postgres DB, or a filesystem path) and attach them to “schedules” and “storage policies”. Everything runs under the Shield service – no external cron needed [19]. It also supports **backup retention** (pruning old archives after X days) and flexible scheduling intervals.	**Encryption:** Shield automatically encrypts backup archives before sending to storage (using a generated key per archive, encrypted with a master key) [20]. **Storage:** Supports many backends – local FS, S3/ MinIO, Azure Blob, WebDAV, etc. [20]. (In an offline setup, one could use a local filesystem or a MinIO service for backup storage).	Shield can be configured to do test restores. Typically, one can set up a “restore target” (for example, a scratch database) and use Shield to perform a restore operation regularly to verify backup integrity. At minimum, logs from each backup job indicate success, and archives can be manually tested. There isn’t a one-click automated sandbox restore unless scripted through the Shield API.	Higher overhead than simple tools. Shield runs as a service (with a web UI and an internal database for job configs). It’s relatively **self-hosted** (there is a Docker Compose to run it). Once set up, it’s quite hands-off: scheduling, notifications, and even a visual dashboard are provided. But initial configuration is complex (defining all targets/ plugins). Maintenance involves monitoring the Shield service itself.	W to Sh lik re na (V Sh de BC Cl ar us Ku CF ac no di ov w di (B by ju da co in sy or lo po w de Ve ar de Sh

Solution	Backup Mechanism & Scope	Automation & Scheduling	Encryption & Storage	Restore Verification	Overhead & Maintenance	K8 Pa
	part of filesystem backup of /etc or via a custom script.					

Solution	Backup Mechanism & Scope	Automation & Scheduling	Encryption & Storage	Restore Verification	Overhead & Maintenance	K8 Pa
Offen's Docker Volume Backup	A simple container that **archives Docker volumes** to a backup file. It runs as a sidecar next to the service container. At backup time, it can **optionally stop the target container** (to quiesce writes) then tar up the volume and resume the container [21] [22]. This yields a full `.tar.gz` of the volume data (e.g. database files). Works per-container; we would run one instance per data volume (Postgres data dir, Neo4j store, etc.).	Scheduling is built-in via a cron expression (set via `BACKUP_CRON_EXPRESSION` env) [23]. Each backup container handles its schedule independently. It also supports retention policy (e.g. keep last N days backups via `BACKUP_RETENTION_DAYS`) [23]. No central view – each sidecar logs its actions.	**Encryption:** Supports optional GPG encryption of the tar archives [24]. **Storage:** Can send archives to local path or cloud: supports S3/ MinIO, WebDAV, Azure, Dropbox, or SSH out of the box [25] [24]. In offline mode, one could point it to a mounted drive or local MinIO service.	Verification is manual – essentially ensuring you can open the tar and perhaps performing a test container restore from it. Offen's tool doesn't auto-test restores. It does provide logs and can send notifications on failures [24], so we know if a backup didn't complete. Integrity is largely based on tar and (optionally) GPG checks.	Very low footprint and simple to deploy (just attach the sidecar to each service in docker-compose with a few env vars). Minimal ongoing maintenance – it's fire-and-forget for each container. However, because it makes full backups every time (no deduplication), storage and time costs can grow for large datasets [26]. Over time, managing many tar files and doing full restores could become slow. Observability is limited (reliant on checking logs or failure notifications).	Th *no* ap Ku (w w Ve sn in si m ga to co la th in ba dr M w dr ap fa or ba Cr Th sn m th co ba tr Ku ba ne

Solution	Backup Mechanism & Scope	Automation & Scheduling	Encryption & Storage	Restore Verification	Overhead & Maintenance	K8 Pa
Bacula/ Bareos	Traditional enterprise backup suite. Uses a client-server model: an agent on the host can dump files, databases (via scripts), etc., to a central backup server. Can certainly back up all required data (with custom pre/post scripts for Neo4j, Qdrant, etc.). Supports full, incremental, differential backups at the file level.	Very feature-rich scheduler: jobs defined in config (text files) with cron-like schedules. Bacula's Director service orchestrates when each job runs and manages retention (Volume recycling). Good for complex policies, but configuration is fairly involved (steep learning curve).	**Encryption:** Supported (Bacula can encrypt backups with TLS or GPG). **Storage:** Many options – disk, tape, cloud via plugins. Could write to local disk or even S3 via gateway. Deduplication not native (enterprise versions have it), compression is supported.	Bacula can do automated verify jobs (it can verify that files in backup match those on disk or do restoration drills). This requires setting up verify schedules and often a separate test restore location. It's doable but adds complexity.	High overhead and complexity for a single-node deployment. You'd need to maintain the Director service, a database for catalog, and possibly clients on each container (or on host mounting volumes). It's battle-tested and reliable, but requires significant admin effort (not "low intervention"). Typically overkill for small environments.	Ba Ku aw co us in or w in m er pr sp op us no lik w to a do co su no st

Notes: Other tools were also considered but deemed less ideal. **Duplicati**, for example, provides a friendly UI and encryption/deduplication, but it has a history of reliability issues (reports of corrupted backup sets and difficult restores) [27] . **Duplicacy** offers efficient lock-free deduplication, but its licensing is not fully open-source for business use [28] . These factors make them less suitable for Janus's needs compared to the options above.

Recommended Solution and Roadmap

After thorough evaluation, **we recommend using BorgBackup with the Borgmatic orchestrator as the backup solution in the current single-node phase**, while preparing for a transition to **Velero (with restic or CSI snapshots)** when the system moves to Kubernetes. This approach best balances our requirements for security, offline operation, low maintenance, and future-proofing. Below we outline the implementation plan and how it addresses Janus's needs:

Phase 1: Implement BorgBackup + Borgmatic on Single Node

Why Borgmatic? Borgmatic is a lightweight wrapper that makes BorgBackup almost turnkey for our scenario. It allows us to define each backup job declaratively in YAML – including what to back up, how often, and any pre/post commands [2]. Crucially, it has built-in support for dumping databases like PostgreSQL and MySQL as part of the backup process [1], and we can leverage its hook system for other services. The benefits at this stage include:

- **Encryption & Deduplication:** All data will be encrypted client-side and deduplicated by Borg. This ensures efficient storage and confidentiality. (Backups are unusable without the encryption key/passphrase.)
- **Per-Service Strategies:** We will configure Borgmatic to run custom scripts before each backup to handle service-specific needs:
- **PostgreSQL:** Use Borgmatic's `postgresql_databases` setting to run `pg_dumpall` or `pg_dump` for specified databases [1]. The dumps will be saved to a dump directory that is included in the backup set. This avoids stopping the Postgres container and ensures a consistent snapshot of the DB.
- **Neo4j:** Because Neo4j Community lacks hot backup, we'll schedule a short downtime at an off-hour. A pre-backup hook (shell script) will gracefully stop the Neo4j container, or utilize `neo4j-admin dump` if available, then restart it after the backup. The raw database files (or dump file) will be captured by Borg.
- **Qdrant:** Use Qdrant's snapshot API to create a snapshot file of each collection [29] [30]. A pre-backup script can call the API (via `curl` or the Python client) to generate snapshots in a known directory. Borg will then back up those snapshot files. This avoids taking Qdrant offline while still getting a consistent backup.
- **MinIO:** The object store can be backed up at the filesystem level (since it's essentially a directory on disk). We will schedule this at a time of low activity, and optionally use a MinIO provided tool (like `mc mirror` to an external drive) as a secondary method. Borg will back up the MinIO data directory; with deduplication, daily backups of largely static objects are space-efficient.
- **Redis:** For Redis with AOF, we can simply back up the AOF file (and RDB file if snapshotting is enabled). We'll trigger a Redis `BGSAVE` or AOF rewrite (BGREWRITEAOF) before the backup to minimize in-memory data. This can be done via a pre-backup Redis CLI command. The Redis container does not need to stop.

- **RabbitMQ:** We will back up RabbitMQ's configuration file(s) and any exported definitions. RabbitMQ allows exporting its definitions (users, queues, etc.) to a JSON file. A periodic export (via `rabbitmqadmin`) can be run and the output included in the backup. The live message queues are not in scope (assumed ephemeral or mirrored); we focus on the config for disaster recovery.

- **Scheduling & Retention:** We'll use systemd timers or cron jobs to run Borgmatic on a schedule (e.g. nightly for most data, possibly more frequently for critical DBs if needed). Each Borgmatic configuration can correspond to a set of services or we can use one config for all with multiple source paths. Retention policies (keep daily/weekly/monthly backups) will be set according to requirements (for example, 7 daily, 4 weekly, 12 monthly) [31]. Borgmatic will automatically prune older backups after each run according to these rules.

- **Storage of Repositories:** Initially, backups will be stored on a **local external drive** or a mounted NAS path (for offline-first reliability). Borg repositories could be on, say, `/backups/janus.borg` on an encrypted volume. To satisfy the requirement of optional sync to object storage, we have two approaches:
- **Periodic Sync to MinIO:** We can configure a daily job (using `rclone` or the MinIO client `mc`) to synchronize the new backup archives to a MinIO bucket. Another approach is to use Borg's `repository` mirroring or use `borg export` to push data to an off-site location. Although Borg doesn't natively speak S3 [32], these workarounds will let us maintain an offsite copy in MinIO.
- **Switch to a Remote Repo:** Alternatively, we could run a tiny SSH service or rest-server on the same host as MinIO and have Borgmatic back up directly to that (simulating an S3 target). Given the complexity, the simpler route is the sync. Restic would have had an advantage here with direct S3 support, but we consider the trade-offs acceptable for Borg's other benefits.
- **Verification & Testing:** Borgmatic will be configured to run `borg check` on the repository periodically (e.g. weekly) [33] to verify integrity of archives and the repo index. Additionally, we will implement a **quarterly test restore** drill: spin up a throwaway VM (or use the same server in a sandbox directory) where we perform a full restore of a random snapshot of each service. For example, restore the Postgres dump and attempt to load it into a test Postgres container to verify it's not corrupted. This process can be semi-automated with a script and gives us confidence in the backups [34] [35]. (In practice, Borg backups are very reliable, but this fulfills the automated restore verification requirement).
- **Monitoring & Alerting:** We will integrate Borgmatic with an alerting mechanism. Borgmatic natively supports Healthchecks.io and similar services to ping on job completion [5]. We'll set up a healthcheck that expects a ping after each backup job – if a backup fails or doesn't run, it will alert us (email/Slack, etc.). Borgmatic can also send logs to syslog or JSON, which we can aggregate. If desired, we could push backup metrics (like last backup age) into our monitoring system (Prometheus) – or simpler, use a tool like **Uptime Kuma or Cronitor** which Borgmatic supports [6]. This ensures we have observability into the backup process (e.g., we'll know if a backup took unusually long or encountered errors).

Maintenance: Once this is set up, ongoing effort is low. Adding a new service to backup means adding a line in the Borgmatic config and possibly a hook script. Borgmatic's declarative config and simple upgrade path (apt/pip update) make it easy to maintain. The backup encryption keys and config will be stored securely (with offline copies of keys as well). We should document the restore procedure for each service so that in a disaster scenario, it's clear how to reconstruct Janus from the Borg repository.

Phase 2: Intermediate Scaling (Optional)

If Janus scales to a small cluster or multiple nodes *before* a full Kubernetes migration (for example, active-active nodes or separate DB and app servers), we can still use the Borgmatic approach:

- **Multiple Nodes:** We could deploy Borgmatic on each node to back up local data stores, or centralize backups by having one node connect to others (over SSH or by mounting volumes over NFS). Borg repositories can accept data from multiple clients, but for simplicity, we might give each node its own repository in this phase to avoid network dependencies. Each node's Borgmatic would be configured similarly.
- **Shield Alternative:** If the infrastructure grows more complex and managing many Borgmatic configs becomes cumbersome, we might revisit a tool like **SHIELD** as an intermediate step. SHIELD could run on one node and coordinate backups across the network (with lightweight agents). This

would provide a single pane for all backups. However, this adds complexity and a database for SHIELD itself, so we'd likely stick with Borgmatic unless absolutely necessary.

Phase 2 is optional and depends on how soon Kubernetes is adopted. Borgmatic can scale modestly well (it's used in many multi-server setups via identical configs on each machine or central NFS mounts).

Phase 3: Kubernetes Migration and Velero Integration

When Janus transitions to Kubernetes, we will migrate the backup strategy to use Kubernetes-native tools, primarily **Velero** for cluster-wide backups and CSI snapshotting for persistent volumes:

- **Velero for Cluster Resources:** Velero is an open-source tool specifically for K8s backup/restore [36]. We will deploy Velero in the cluster, configured to use a MinIO bucket as its backup store (Velero supports any S3-compatible storage). Velero will back up Kubernetes resources (config maps, secrets, deployments, etc.) as well as volume data. This covers RabbitMQ config (which by then might be stored as a ConfigMap or a persistent volume) and any other statefulset configurations.
- **Volume Backups with Restic/Kopia:** For the actual data in volumes (Postgres PVCs, Neo4j PVCs, etc.), we have two options:
- Use Velero's **restic integration** (stable, beta feature) to backup PV file systems to the object store [37]. Velero will automatically coordinate with restic DaemonSet pods on each node to snapshot the files in the volumes. This is encrypted and deduplicated (restic repository chunks stored in the MinIO bucket). We would ensure Velero's restic repo is initialized with encryption keys (Velero handles this per cluster).
- Alternatively, by the time of migration, Velero may fully support **Kopia integration** as indicated in newer releases [17]. Kopia could offer performance benefits. We'll choose whichever is mature at that point. (Restic is proven; Kopia is emerging with better speed).
  In either case, Velero will effectively replace our manual Borgmatic backups for the data volumes. We expect to schedule Velero backups similarly (e.g., nightly) using Velero's Schedule CRDs [38].
- **Database Considerations on K8s:** In Kubernetes, an alternative approach is to continue performing logical dumps inside the pods (especially for Postgres and Neo4j) and backing those up, instead of relying only on volume snapshots. This can be done via Kubernetes CronJobs:
- For example, run a daily CronJob that execs `pg_dumpall` in the Postgres pod to produce a dump file in an `archives` volume, which Velero then backs up. This provides an extra layer of backup (both a file-level backup and a logical backup). Velero supports **backup hooks** [39] which we can use to quiesce databases or trigger dumps before taking the snapshot of the volume.
- We will leverage hooks: e.g., configure Velero to run a pre-backup hook on the Neo4j pod to invoke a script that flushes data or creates a backup file, so that the snapshot contains consistent data [39].
- **CSI Snapshots (if available):** If the Kubernetes storage class supports CSI snapshots, we might use Velero's CSI integration for more efficient volume snapshots (especially for large volumes). This could complement or replace file-level backups for certain databases, providing near-instant snapshot points that can be backed up off-cluster. CSI snapshots would be stored in the storage system and could be moved off-cluster via Velero's upload (dependent on driver capabilities).
- **Retention & Offsite:** We'll mirror the retention policies in Velero (it can be configured to prune old backups). The MinIO bucket with Velero data can be the same as used in Phase 1, or a new bucket – as long as we manage capacity. Old Borg archives could even be imported manually if needed, but likely we'll start fresh on K8s. We will also ensure offsite copies of the Velero backup store (e.g., by setting up MinIO bucket replication to an external drive or another site).

- **Sunset Phase 1 Backups:** During the transition, we might run both systems in parallel for a while (e.g., keep Borgmatic running until the K8s backups are confirmed working). Eventually, Borgmatic on the old node can be decommissioned once all data resides in the cluster and is handled by Velero.

Observability in K8s: Velero comes with built-in monitoring facets – it logs events to the cluster, and we can collect metrics on backup success/failure. We will integrate Velero's metrics (via Prometheus scraping or the Velero CLI) to ensure backups continue to run regularly. Additionally, test restores will be conducted in a staging cluster to ensure we can recover from Velero backups as part of DR drills.

Long-Term Maintenance and Observability

Over the long term, the chosen strategy (Borgmatic now, Velero later) aligns with **low maintenance and high reliability**:

- Borgmatic's configuration-as-code and healthcheck integration mean we'll get quick feedback if something breaks [5]. The team won't need to constantly babysit backups – just check alerts and periodically review logs or test restores.
- The backup processes are fully automated on schedule, and new services can be onboarded with a small config change. This declarative approach makes it easy to track changes (we can keep the Borgmatic config YAML in version control).
- In an offline scenario, everything runs locally. In the event of a total failure of the main node, we have encrypted backup data on an external medium that can be used to restore on a new node. Since the backups are encrypted, losing the physical media doesn't expose data. We will just need the Borg passphrase (which will be stored securely offline as well).
- **Disaster Recovery process:** We will document a runbook for restoring the entire Janus stack from backups. E.g., "If node dies: install Docker and Borg, retrieve backup drive, run `borg extract` to recover volumes, spin up containers, import dumps, etc." Having this written and periodically tested is the ultimate assurance that our backups are effective.

By moving to Velero on Kubernetes, we'll gain even better integration: Velero will handle both app data and cluster config, making full cluster recovery one streamlined process. Velero's hooks will allow us to carry over any special handling (like quiescing databases) in a Kubernetes-friendly way [39]. At that stage, **observability** improves further with Kubernetes's ecosystem – we can use tools like Grafana dashboards for Velero or even look into vendors (if needed) that specialize in K8s backup visibility. However, our plan emphasizes sticking with open-source, self-hosted solutions to maintain the "self-reliant" philosophy.

Conclusion

In summary, our recommended backup stack for Janus is a combination of **Borgmatic-orchestrated encrypted backups** in the short term, transitioning to **Velero-based cluster backups** in the long term. This approach meets all the project's goals:

- It covers all data sources (through a mix of **native tools and hooks** for each service).
- It operates in an offline environment and uses strong encryption (client-side encryption in Borg ensures data security [40]).
- It is self-hosted and lightweight – essentially just an extra container (or cron job) on the host running Borgmatic, with no heavy infrastructure needed.

- It supports automated verification by leveraging Borg's integrity checks and periodic restore tests.
- It requires minimal manual effort day-to-day; jobs are defined declaratively and run automatically, with alerting on failure.
- It's Docker-friendly (no dependence on Kubernetes or cloud services at this phase) yet provides a clear pathway to a **Kubernetes-native backup regimen** (Velero with restic/Kopia) when we scale up [36].

By exploring newer solutions like Backrest and SHIELD, we ensured our choice wasn't just by habit but truly the best fit. Those alternatives have merits (e.g. Backrest's UI or SHIELD's all-in-one design), but Borgmatic hits the sweet spot for Janus's current scale and team preferences – it's simple, robust, and widely trusted in the community. As we grow, the investment in a solid backup foundation now will pay off, and we'll be well-prepared to evolve our strategy alongside Janus's infrastructure. With these measures in place, the Janus backend will be safeguarded against data loss and ready for quick recovery from any disaster – all while keeping maintenance overhead low and confidence high.

Sources:

- Borgmatic official docs – configuration and features [41] [1]
- Nic Raboy, *Easy Automated Docker Volume Backups* – comparison of Offen vs Backrest (container stop for consistency, incremental backups with Restic) [8] [42]
- Backrest project – web UI and scheduling on top of Restic [7] [43]
- SHIELD project – backup orchestration with plugins for Postgres, Redis, etc., and encrypted storage support [19] [20]
- Eric Cheng's backup article – evaluation of restic, Borg, Kopia (performance and features) [16] [44]
- Velero official site – Kubernetes backup/restore features and hook capabilities [36] [45]
- Bacula Systems blog – notes on Docker volume backup methods and Offen's capabilities (GPG encryption, rotation, notifications) [24]

---

[1] [2] [3] [4] [5] [6] [31] [33] [40] [41] borgmatic
https://torsion.org/borgmatic/

[7] [9] [10] Backrest: a cross platform backup orchestrator and WebUI for restic - Community - restic forum
https://forum.restic.net/t/backrest-a-cross-platform-backup-orchestrator-and-webui-for-restic/7069

[8] [21] [22] [23] [26] [42] Easy Automated Docker Volume Backups That Are Database Friendly
https://www.thepolyglotdeveloper.com/2025/05/easy-automated-docker-volume-backups-database-friendly/

[11] [12] [43] Automated Home Server Backups Made Easy: Restic, Backrest, and Backblaze B2
https://berkertopuz.com/automated-home-server-backups-made-easy-restic-backrest-and-backblaze-b2/

[13] [14] [15] [16] [27] [28] [32] [34] [35] [44] Backups: the good, the bad, and the ugly · Eric Cheng
https://www.chengeric.com/backups/

[17] Velero Docs - Velero File System Backup Performance Guide
https://velero.io/docs/v1.15/performance-guidance/

[18] [19] [20] GitHub - shieldproject/shield: A standalone system that can perform backup and restore functions for a wide variety of pluggable data systems
https://github.com/shieldproject/shield

24 How to backup Docker containers? Docker volume backup in 2025
https://www.baculasystems.com/blog/docker-backup-containers/

25 Automate backing up your Docker volumes - DEV Community
https://dev.to/offen-software/automate-backing-up-your-docker-volumes-3gdk

29 30 Snapshots - Qdrant
https://qdrant.tech/documentation/concepts/snapshots/

36 37 38 39 45 Velero
https://velero.io/