Cloud Workspace Architecture for Autonomous Agent Systems on Kubernetes

1. Context and Architectural Paradigm Shift

The deployment of autonomous Artificial Intelligence (AI) agents within a Kubernetes environment represents a fundamental paradigm shift from traditional microservice architectures. While standard microservices are typically stateless, idempotent, and horizontally scalable, autonomous agents—specifically those employing architectures like LangGraph or ReAct—exhibit behaviors that are inherently stateful, ephemeral, and computationally distinct. This report provides an exhaustive analysis of the architectural requirements for deploying a LangGraph-based agent system to a k3s Kubernetes cluster, specifically addressing the unique challenges of isolation, persistence, and inter-process communication in a high-capability homelab environment that retains enterprise scalability.

The transition from a monolithic "agent server" to a distributed, containerized fleet necessitates a re-evaluation of standard Kubernetes primitives. An agent operating in a "phase-alternating state machine" (cycling between strategic reasoning and tactical execution) is not merely a service waiting for requests; it is a computational job with a distinct lifecycle, a complex internal state, and a requirement for a persistent, mutable filesystem "workspace." This workspace acts as the agent's short-term memory, housing cloned repositories, intermediate code artifacts, and log files. The ephemeral nature of these tasks—spinning up to solve a problem and tearing down upon completion—clashes with the "always-on" philosophy of standard Kubernetes Deployments, requiring a more nuanced approach to workload management.[1]

Furthermore, the integration of Large Language Models (LLMs) introduces a layer of non-determinism and security risk that is absent in deterministic software. Agents are effectively remote code execution (RCE) engines by design; their utility is derived from their ability to generate and execute code. This creates a tension between the need for broad tool access (to be useful) and strict isolation (to be secure).[3] The architecture proposed herein addresses this tension through a "defense-in-depth" strategy, leveraging Kubernetes-native constructs to sandbox agent behaviors while maintaining the high-performance I/O required for rapid reasoning loops.

2. Container Lifecycle Patterns and Workload Management

The selection of the appropriate Kubernetes workload controller is the foundational decision that dictates networking, storage binding, and cleanup logic. The agentic workflow described implies a workload that is transient but stateful during its execution window.

2.1 Comparative Analysis of Workload Primitives

The Kubernetes ecosystem offers several primitives for managing pod lifecycles: **Deployments**, **StatefulSets**, and **Jobs**. Each offers distinct guarantees regarding restart policies, identity stability, and completion logic.

The Limitations of Deployments for Agentic Workloads

Deployments are the standard for stateless services (e.g., REST APIs). They are designed to maintain a specified number of replicas continuously. If a pod within a Deployment terminates successfully (exit code 0), the Deployment controller interprets this as a failure to maintain availability and immediately restarts the pod.[5] For an agent task that has a defined end state (e.g., "Refactor completed"), this behavior is counter-productive. It forces the external Orchestrator to actively monitor the agent's internal state and explicitly scale the Deployment to zero to stop execution. This introduces a "split-brain" risk where the Orchestrator believes a task is complete, but the Kubernetes controller continues to burn resources by restarting the finished agent.[2]

The Strategic Fit of Kubernetes Jobs

The **Kubernetes Job** resource is semantically aligned with the agent's lifecycle. A Job creates one or more pods and ensures that a specified number of them successfully terminate.

- **Lifecycle Alignment:** When an agent completes its objective and exits, the Job controller marks the task as Complete. This native handling of "completion" eliminates the need for the Orchestrator to intervene to stop the process, reducing control plane chatter and potential race conditions.[1]
- **Error Handling and Backoff:** Agents interacting with external LLM APIs (OpenAI, Anthropic) are subject to transient network failures or rate limits. A Job supports the backoffLimit field, allowing the system to retry a failed pod a set number of times with an exponential backoff delay. This creates a robust mechanism for distinguishing between transient infrastructure glitches and fatal logical errors in the agent's code.[1]
- **Resource Cleanup:** The ttlSecondsAfterFinished field in the Job specification provides a native mechanism for cascading deletion. Once a job completes (either Complete or Failed), the control plane can automatically delete the Job and its dependent Pods after a configured interval.[6] This feature is critical for preventing resource leaks in a system that might spawn thousands of ephemeral agents over time.

Custom Resource Definitions (CRDs) and Operators

For enterprise-scale deployments, the "Gold Standard" is often a Custom Resource Definition

(CRD), such as an AgentTask, managed by a custom Operator.[7] An Operator can encode complex domain logic, such as "snapshot the workspace to S3 upon failure" or "keep the pod alive for debugging if the agent output contains specific error keywords".[3]

However, for a k3s homelab environment, the maintenance burden of writing and updating a custom Operator is significant. The native **Job** primitive, when combined with a smart Orchestrator, offers 80% of the benefit with minimal operational overhead. The Orchestrator can simply template a Job manifest and apply it, relying on standard Kubernetes behaviors.[9]

Recommendation: Utilize **Kubernetes Jobs** for the agent execution. The Orchestrator should act as a factory, generating a unique Job manifest for each task uuid. This offloads the heavy lifting of restart policies and completion tracking to the Kubernetes controller manager, allowing the Orchestrator to focus on higher-level task queue management.

2.2 Managing Long-Running Processes

The user query raises a valid concern regarding "long-running jobs (hours)." In standard batch processing, Jobs are often short-lived. However, Kubernetes Jobs have no inherent time limit unless activeDeadlineSeconds is specified.[1] An agent performing a complex refactor or deep research task may run for several hours.

To support robust long-running sessions, the architecture must account for node failures and "zombie" processes:

1. **Liveness Probes:** Even though the workload is a Job, livenessProbes should be configured. If an agent enters a deadlock state (e.g., waiting indefinitely for an LLM response that was dropped), the probe will fail, prompting Kubernetes to kill the container. Because the workload is a Job, this kill event is treated as a pod failure, triggering a replacement pod (up to the backoffLimit).[10]
2. **Checkpointing Persistence:** Since a Job might be restarted on a different node, the agent's state must be persisted externally. The system's reliance on SQLite checkpoints (workspace/checkpoints/job_{id}.db) is the mechanism for this durability.[2] When a replacement pod starts, it must check for the existence of this checkpoint file and resume execution from the last known state, rather than restarting the reasoning loop from scratch.

2.3 Sidecar Patterns for Observability

In a distributed agent system, the logs are not merely operational telemetry; they are the primary product consumed by the user (the "thought trace"). Relying on kubectl logs is insufficient for a production-grade system because it couples log retrieval to the availability of the Kubernetes API server and the existence of the pod.[12]

The recommended pattern is a **Sidecar Container** running a lightweight log shipper like

Fluent Bit or **Vector**.[13]

- **Architecture:** The main agent container writes its logs to a shared emptyDir volume (e.g., /var/log/agent/agent.log) or a named pipe. The sidecar container mounts the same volume, tails the file, and streams the logs to the Orchestrator or a centralized logging backend (e.g., Loki, OpenObserve).[14]
- **Benefits:** This decouples log shipping from the agent's runtime logic. Crucially, if the agent container crashes due to an Out-Of-Memory (OOM) error, the sidecar remains running long enough to ship the final stack trace and "last gasp" logs before the Pod is terminated. This ensures that the reason for failure is captured, which is often lost in simple stdout buffering scenarios.[16]

3. Storage Architecture and Workspace Persistence

The requirement for **SQLite** persistence coupled with a distributed Kubernetes environment presents the most significant architectural constraint. SQLite is fundamentally a file-based database that relies heavily on POSIX advisory locks to manage concurrency. In a clustered environment, the choice of storage backend dictates not just performance, but data integrity.

3.1 The Concurrency and Locking Dilemma

SQLite's Write-Ahead Logging (WAL) mode is essential for the performance of high-throughput agent loops. WAL mode allows for significantly higher concurrency than the legacy rollback journal mode, as readers do not block writers.[17] However, WAL mode relies on a shared memory file (.shm) that is mapped into the process's memory space using mmap.

The Failure of NFS: Standard Network File System (NFS) implementations are fundamentally unsuited for SQLite WAL mode. NFS often has high latency for file locking operations and, more critically, creates inconsistencies in the memory-mapped .shm file across different clients (or even the kernel's view of the file).[19] Using SQLite over NFS frequently results in database is locked errors or database corruption.[20] Snippet [22] explicitly notes that "WAL does not work over a network filesystem" due to these limitations.

The Limitations of Distributed Block Storage (Longhorn/Ceph): Solutions like Longhorn provide block storage (iSCSI), which presents a block device to the node OS. While this supports local filesystem semantics (like ext4 or xfs) on top of the block device, mitigating the locking issues of NFS, it introduces significant I/O latency. Longhorn performs synchronous replication to ensure data durability; every write operation must be committed to multiple replicas across the network before it is acknowledged.[23] For an agent workload that is write-heavy (constantly updating "thoughts," "todos," and creating small temporary files), this latency is punishing. Benchmarks indicate that Longhorn can be 5x to 10x slower than local storage for random write operations, which characterizes SQLite traffic.[23]

3.2 Recommended Solution: Tiered Local Storage

For the targeted k3s homelab environment, prioritizing performance and simplicity is key. The **Local Path Provisioner** is the recommended primary storage mechanism, supplemented by an object storage archiving strategy.

Tier 1: Active Workspace (Local Path Provisioner)

The rancher.io/local-path storage class comes bundled with k3s. It dynamically provisions a Persistent Volume (PV) backed by a directory on the host node's local filesystem (NVMe or SSD).[23]

- **Performance:** This offers native I/O speeds, essential for SQLite WAL operations and rapid file manipulations (e.g., git clone, grep, pip install). It eliminates the network latency overhead of iSCSI or NFS.
- **Simplicity:** It requires zero configuration and consumes negligible system resources compared to the heavy controller pods required by Longhorn or Rook/Ceph.[21]

Constraint: The data is node-bound. If the node hosting the pod fails, the data is inaccessible until the node recovers. However, for *ephemeral* agent tasks, this is an acceptable trade-off, provided there is a backup mechanism.

Tier 2: Durable Archival (MinIO/S3)

To mitigate the risk of data loss from node failure and to support the "stateless between jobs" requirement, the architecture should treat the Local Path PVC as a high-speed scratchpad.

1. **Initialization:** An InitContainer is used to pre-populate the workspace. It checks if a remote backup exists (e.g., in MinIO) for the given Job ID and downloads it to the /workspace volume before the agent starts.[25]
2. **Continuous Snapshotting:** A sidecar container or a background thread within the agent application periodically (e.g., every 5 minutes) creates a snapshot of the checkpoints.db and syncs it to the S3 bucket. This ensures that if the node dies, the work lost is minimal.
3. **Finalization:** Upon successful completion, the final state of the workspace is uploaded to S3/MinIO for long-term storage, and the local PVC can be discarded.

3.3 Dynamic PVC Lifecycle and Cleanup

To maintain strict isolation, each Job must utilize a dedicated **PersistentVolumeClaim (PVC)**. Sharing a single PVC with subdirectories violates the isolation principle and creates security risks where one agent might traverse directories to access another agent's data.[26]

Automated Cleanup via OwnerReferences: A major challenge with dynamic PVCs in Kubernetes is that they are not automatically deleted when the Pod or Job that used them is deleted.[27] This can lead to "orphaned" volumes consuming all disk space. The solution is to utilize Kubernetes **Garbage Collection** by setting ownerReferences on the PVC. When the

Orchestrator creates the PVC, it should set the ownerReferences field to point to the associated Job.

- **Mechanism:** When the Job's ttlSecondsAfterFinished expires and the Job is deleted by the control plane, the Garbage Collector identifies the PVC as a dependent resource and cascades the deletion, removing the PVC and the underlying data.[28]

Manifest Configuration:

YAML

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: workspace-job-123
  namespace: agents
  ownerReferences:
    - apiVersion: batch/v1
      kind: Job
      name: agent-job-123
      uid: <UUID-of-Job>  # Injected by Orchestrator
      controller: true
      blockOwnerDeletion: true
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: local-path
  resources:
    requests:
      storage: 5Gi
```

Note: Since the Job UID is not known until the Job is created, the Orchestrator must create the Job first, retrieve its UID, and then create the PVC with the correct reference, or use a pre-determined deterministic UID generation strategy if supported.

4. Communication Pattern: Orchestrator ↔ Agent

The current "REST API per pod" architecture described in the prompt is brittle in a dynamic Kubernetes environment. It relies on the Orchestrator discovering and tracking the IP addresses of ephemeral pods, which change constantly. It also forces the agent to act as a

server, complicating its lifecycle management (e.g., handling connection draining).

4.1 Transition to an Event-Driven Push Architecture

The recommended pattern is to invert the communication model: instead of the Orchestrator *pulling* data from the Agent, the Agent *pushes* data to the Orchestrator via a message broker. **NATS** is the recommended technology for this layer due to its lightweight nature, high performance, and "cloud-native" design.[30]

The NATS Advantage

Compared to Redis, NATS (specifically NATS JetStream) offers superior support for "at-least-once" delivery and subject-based routing, which maps well to the topic-based nature of agent communication.[30]

- **Command Plane:** The Orchestrator publishes tasks to a subject like cmd.agent.<job_id>. The Agent subscribes to this subject upon startup. This completely decouples network identity; the Orchestrator does not need to know the Pod IP, only the Job ID.[31]
- **Control Plane:** Status updates (e.g., "Phase Changed to TACTICAL") are published to events.agent.<job_id>.
- **Data Plane:** Logs and streaming text are published to logs.agent.<job_id>.

4.2 Real-Time Log Streaming to Cockpit

To expose these logs to the Angular Cockpit in real-time, the Orchestrator (FastAPI) acts as a bridge between the NATS bus and the frontend using **Server-Sent Events (SSE)**.

Architecture Flow:

1. **Log Emission:** The Python agent uses a custom logging.Handler that formats log records as JSON and publishes them to the NATS subject logs.agent.<job_id>.
2. **Subscription:** The FastAPI Orchestrator exposes an endpoint /api/stream/{job_id}. When the Angular frontend connects to this endpoint, the backend creates a NATS subscription to the corresponding subject.
3. **Streaming:** As messages arrive on the NATS bus, the FastAPI handler yields them to the HTTP client using EventSourceResponse (from sse-starlette). This allows the frontend to receive a continuous stream of text and state updates without polling.[33]

This pattern is highly scalable. Even if the agent pod crashes and is rescheduled, the new pod simply starts publishing to the same NATS subject, and the frontend stream continues uninterrupted (perhaps with a "Reconnecting..." status message pushed by the sidecar).

5. Tool Architecture and the Execution Runtime

The execute(command: str) tool represents the most powerful capability of an autonomous agent, effectively granting it shell access. While specialized tools (like read_file) are safer, a

general shell tool is often necessary for dynamic tasks like running tests or linting code.

5.1 The execute() Design Principles

Recommendation: Implement execute(), but isolate its scope.

Standardizing on a general execution tool avoids the need to wrap every possible shell command (ls, grep, find, cat) into a Python function. However, this power must be constrained. The execute tool should not run strictly as a subprocess of the main agent logic if possible, but for simplicity in a k3s homelab, running it as a subprocess is acceptable *if* the environment is hardened.

5.2 Managing Dependencies in Read-Only Containers

Security best practices dictate that the container's root filesystem should be mounted as **Read-Only** (readOnlyRootFilesystem: true).[36] This prevents the agent (or a compromised process) from modifying system binaries or establishing persistence via rootkits.

However, this breaks standard workflows like pip install or git clone, which expect to write to /usr/local/lib or the current working directory.

The Ephemeral Overlay Solution: The solution is to mount writable emptyDir volumes at specific paths where write access is legitimate.[38]

1. **User-Space Package Installation:**
    - Set the environment variable PYTHONUSERBASE=/home/agent/.local.
    - Update PATH=$PATH:/home/agent/.local/bin.
    - Mount an emptyDir volume to /home/agent/.local.
    - When the agent runs pip install <package>, it will default (or can be forced via --user) to install into the writable volume. This allows the agent to install dependencies for the current task without modifying the system image.
2. **Workspace Operations:**
    - The WORKSPACE_PATH (e.g., /workspace) is backed by the Local Path PVC (Tier 1 storage). This is writable.
    - All git clone, code generation, and file manipulation occur strictly within this mount point. The execute tool should enforce that the working directory is always within /workspace.

5.3 Is /dev/shm Needed?

Yes. As noted in Section 3.1, SQLite WAL mode requires shared memory. In a Docker/Kubernetes container, /dev/shm is typically limited to 64MB by default. For heavy SQLite usage, this may be insufficient. **Recommendation:** Mount a Memory backed emptyDir volume to /dev/shm. This allows the shared memory segment to grow as needed (up to the

pod's memory limit), preventing SQLite crashes during large transactions.[40]

6. Security and Isolation Model

Running autonomous agents is effectively running "Remote Code Execution as a Service." The security model must assume the agent will try to escape.

6.1 Network Policy: The Egress Allowlist Problem

Agents typically need access to specific external resources (e.g., api.openai.com, github.com) but should be blocked from accessing the internal network (e.g., the K8s API server, other pods, the metadata service).

The Challenge: Standard Kubernetes NetworkPolicies operate at Layer 3/4 (IP/Port). They do not support DNS names (FQDNs). Since the IP addresses of GitHub and OpenAI change dynamically via CDN, maintaining static IP allowlists is operationally impossible.[42]

Recommendation: The Proxy Egress Pattern

To achieve FQDN whitelisting without complex enterprise CNIs (like Cilium, which supports toFQDNs but is heavy for k3s), deployment of a transparent **Egress Proxy** is recommended.

1. **Proxy Pod:** Deploy a lightweight proxy (like **Squid** or **TinyProxy**) in the cluster. Configure it with an allowlist of domains (.github.com, .openai.com, pypi.org).
2. **Agent Configuration:** Inject standard proxy environment variables into the Agent pod:
   Bash
   ```
   export HTTP_PROXY="http://proxy-service.default.svc.cluster.local:3128"
   export HTTPS_PROXY="http://proxy-service.default.svc.cluster.local:3128"
   ```
3. **NetworkPolicy Enforcement:** Apply a NetworkPolicy to the Agent pod that:
   - **Denies All Egress** by default.
   - **Allows Egress** only to the Proxy Service (on port 3128) and the kube-dns service (port 53 UDP/TCP).
   - Crucially, block access to the K8s API server IP and the cloud metadata IP (169.254.169.254).[4]

This architecture effectively creates a "Layer 7 Firewall" using standard components manageable in a homelab.

6.2 Secrets Strategy

Secrets (API keys, DB credentials) must be injected into the agent environment.

- **Risk:** If secrets are injected as Environment Variables, a simple execute("env") by the agent reveals them.
- **Mitigation:** Inject secrets as **Files** using a Kubernetes Secret volume mounted at

/var/secrets.

- **Code Level:** The agent code should read the API key from the file to initialize the client and then *immediately* clear the variable from memory if possible. The file path should not be in the agent's "working directory" (i.e., not in /workspace).
- **Redaction:** The Orchestrator's log streaming service should implement a regex filter to redact known secret patterns (like sk-...) before sending logs to the frontend.

6.3 Pod Security Standards (PSS)

The k3s cluster should enforce the **Restricted** Pod Security Standard for the agent namespace.

- **Non-Root:** runAsNonRoot: true. Run as UID 1000.
- **Capabilities:** capabilities: drop: ["ALL"]. The agent does not need NET_ADMIN or SYS_ADMIN.
- **Seccomp:** seccompProfile: type: RuntimeDefault. This blocks dangerous syscalls that are often used for container escapes.[45]

7. Reference Implementations

7.1 OpenDevin (OpenHands)

OpenDevin serves as a primary reference for the "Workspace" pattern. It utilizes a Docker-based sandbox where the workspace is mounted as a volume shared between the agent logic and the execution environment.

- **Relevance:** OpenDevin validates the separation of "Agent State" (memory/LLM context) from "Workspace State" (files/code).
- **Adaptation:** While OpenDevin often uses Docker-in-Docker for local execution, our k3s architecture replaces this with the Kubernetes Job + PVC model, providing better native integration and security.[47]

7.2 Anthropic "Computer Use" / Sandbox Runtime

Anthropic's implementation emphasizes network isolation. Their "Sandbox Runtime" creates a strictly isolated environment where the only egress is via a Unix Domain Socket proxy.

- **Relevance:** This confirms that direct network access for agents is an anti-pattern. Our "Proxy Egress Pattern" (Section 6.1) is the Kubernetes-native equivalent of Anthropic's Unix socket proxy, providing a choke point for auditing and filtering external traffic.[4]

8. Summary of Recommendations

Architectural Component	Recommendation	Rationale

Container Lifecycle	**Kubernetes Job**	Matches the finite, task-based nature of agents; handles retries and cleanup natively.
Persistence	**Local Path Provisioner**	Provides necessary I/O performance for SQLite WAL; simplest for k3s.
Concurrency	**One PVC per Job**	Ensures strict isolation; OwnerReferences handles automated cleanup.
Communication	**NATS + SSE**	Decouples services; enables real-time, event-driven log streaming to UI.
Tooling	**Execute (Bash)**	Essential for flexibility; secured via read-only root & ephemeral emptyDir mounts.
Security	**NetworkPolicy + Squid Proxy**	Solves the FQDN filtering problem; prevents internal network probing.

This architecture delivers a robust, secure, and scalable foundation for running autonomous AI agents. It balances the high-performance requirements of modern "thinking" loops with the strict isolation mandated by executing untrusted code, all while remaining manageable within a k3s homelab context.

Works cited

1. Jobs | Kubernetes, accessed February 1, 2026, https://kubernetes.io/docs/concepts/workloads/controllers/job/
2. Kubernetes for agentic apps: A platform engineering perspective, accessed February 1, 2026, https://platformengineering.org/blog/kubernetes-for-agentic-apps-a-platform-engineering-perspective
3. ADA: Automated Moving Target Defense for AI Workloads via Ephemeral

Infrastructure-Native Rotation in Kubernetes - arXiv, accessed February 1, 2026, https://arxiv.org/html/2505.23805v1
4. Making Claude Code more secure and autonomous with sandboxing - Anthropic, accessed February 1, 2026, https://www.anthropic.com/engineering/claude-code-sandboxing
5. Workloads - Kubernetes, accessed February 1, 2026, https://kubernetes.io/docs/concepts/workloads/
6. Automatic Cleanup for Finished Jobs - Kubernetes, accessed February 1, 2026, https://kubernetes.io/docs/concepts/workloads/controllers/ttlafterfinished/
7. Our experience running an AI workload in Kubernetes — Part 1 | by Jakub Hlavačka | Berops Blog | Medium, accessed February 1, 2026, https://medium.com/berops-blog/our-experience-running-an-ai-workload-in-kubernetes-part-1-16633c206f3f
8. Extend the Kubernetes API with CustomResourceDefinitions, accessed February 1, 2026, https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/
9. Automating the Kubernetes Cleanup with Argo Workflows: Because Even Admins Need a Break | by Dordavid | Medium, accessed February 1, 2026, https://medium.com/@dordavid015/automating-the-kubernetes-cleanup-with-argo-workflows-because-even-admins-need-a-break-eacb1729c048
10. deployment.yaml - All-Hands-AI/OpenHands-Cloud - GitHub, accessed February 1, 2026, https://github.com/All-Hands-AI/OpenHands-Cloud/blob/main/charts/openhands/templates/deployment.yaml
11. Redis Enterprise for Kubernetes architecture | Docs, accessed February 1, 2026, https://redis.io/docs/latest/operate/kubernetes/7.4.6/architecture/
12. Logging Architecture - Kubernetes, accessed February 1, 2026, https://kubernetes.io/docs/concepts/cluster-administration/logging/
13. A Practical Guide to Kubernetes Logging - Logz.io, accessed February 1, 2026, https://logz.io/blog/a-practical-guide-to-kubernetes-logging/
14. 6 Kubernetes Logging Best Practices - vCluster, accessed February 1, 2026, https://www.vcluster.com/blog/6-kubernetes-logging-best-practices
15. Efficient Kubernetes Log Streaming: Real-time Insights for DevOps - OpenObserve, accessed February 1, 2026, https://openobserve.ai/blog/efficient-kubernetes-log-streaming/
16. Kubernetes Logging Best Practices - Graylog, accessed February 1, 2026, https://graylog.org/post/kubernetes-logging-best-practices/
17. SQLite Optimizations For Ultra High-Performance - PowerSync, accessed February 1, 2026, https://www.powersync.com/blog/sqlite-optimizations-for-ultra-high-performance
18. SQLite concurrent writes and "database is locked" errors - Ten thousand meters, accessed February 1, 2026, https://tenthousandmeters.com/blog/sqlite-concurrent-writes-and-database-is-l

ocked-errors/
19. HostPath or NFS? (Longhorn or NFS Provisioner)? PERFORMANCE : r/kubernetes - Reddit, accessed February 1, 2026, https://www.reddit.com/r/kubernetes/comments/15eeudi/hostpath_or_nfs_longhorn_or_nfs_provisioner/
20. K3S SQLite slow on NFS - kubernetes - Stack Overflow, accessed February 1, 2026, https://stackoverflow.com/questions/71122901/k3s-sqlite-slow-on-nfs
21. Best choice for storage in k3s cluster : r/kubernetes - Reddit, accessed February 1, 2026, https://www.reddit.com/r/kubernetes/comments/1b245wx/best_choice_for_storage_in_k3s_cluster/
22. Locking issues on multi user database - SQLite User Forum, accessed February 1, 2026, https://sqlite.org/forum/info/a6675453ecd9af62d13d55fb38562a2e93c434c57f7994b34a4fae91506a3214?t=c
23. Comparing Longhorn vs LocalPath Performance in My Kubernetes Cluster - Medium, accessed February 1, 2026, https://medium.com/@saeed.b.67/comparing-longhorn-vs-localpath-performance-in-my-kubernetes-cluster-2ace7073f534
24. Volumes and Storage - K3s - Lightweight Kubernetes, accessed February 1, 2026, https://docs.k3s.io/add-ons/storage
25. Persistent Storage in Kubernetes: A Comprehensive Guide | by Senthil Raja Chermapandian | Medium, accessed February 1, 2026, https://medium.com/@senthilrch/persistent-storage-in-kubernetes-a-comprehensive-guide-6aeb4016c2a2
26. database is locked errors · Issue #164 · canonical/dqlite - GitHub, accessed February 1, 2026, https://github.com/canonical/dqlite/issues/164
27. Auto delete persistant volume claim when a kubernetes job gets completed - Stack Overflow, accessed February 1, 2026, https://stackoverflow.com/questions/51023650/auto-delete-persistant-volume-claim-when-a-kubernetes-job-gets-completed
28. Auto delete PVC when scaling down? [closed] - Stack Overflow, accessed February 1, 2026, https://stackoverflow.com/questions/51519301/auto-delete-pvc-when-scaling-down
29. Owners and Dependents - Kubernetes, accessed February 1, 2026, https://kubernetes.io/docs/concepts/overview/working-with-objects/owners-dependents/
30. Compare NATS - NATS Docs - NATS.io, accessed February 1, 2026, https://docs.nats.io/nats-concepts/overview/compare-nats
31. NATS Messaging vs REST Performance Comparison - Vinsguru, accessed February 1, 2026, https://blog.vinsguru.com/nats-performance-comparison-with-rest/
32. AI agent orchestration for production systems - Redis, accessed February 1, 2026, https://redis.io/blog/ai-agent-orchestration/

33. Server Sent Events with FastAPI - Medium, accessed February 1, 2026, https://medium.com/@upesh.jindal/server-sent-events-with-fastapi-ab9ed99ccac4
34. Realtime Log Streaming with FastAPI and Server-Sent Events | Amit Tallapragada, accessed February 1, 2026, https://amittallapragada.github.io/docker/fastapi/python/2020/12/23/server-side-events.html
35. Building a Real-time Streaming API with FastAPI and OpenAI: A Comprehensive Guide | by stark | Medium, accessed February 1, 2026, https://medium.com/@shudongai/building-a-real-time-streaming-api-with-fastapi-and-openai-a-comprehensive-guide-cb65b3e686a5
36. Configure a Security Context for a Pod or Container - Kubernetes, accessed February 1, 2026, https://kubernetes.io/docs/tasks/configure-pod-container/security-context/
37. Kubernetes 1.30: Read-only volume mounts can be finally literally read-only, accessed February 1, 2026, https://kubernetes.io/blog/2024/04/23/recursive-read-only-mounts/
38. Installing from a read-only source - Packaging - Discussions on Python.org, accessed February 1, 2026, https://discuss.python.org/t/installing-from-a-read-only-source/4115
39. pip editable install on read-only filesystem - python - Stack Overflow, accessed February 1, 2026, https://stackoverflow.com/questions/38798106/pip-editable-install-on-read-only-filesystem
40. Elastic Container Instance:Mount an emptyDir volume to change the shm size - Alibaba Cloud, accessed February 1, 2026, https://www.alibabacloud.com/help/en/eci/user-guide/mount-an-emptydir-volume-to-modify-the-shm-size-of-a-pod
41. Define size for /dev/shm on container engine - Stack Overflow, accessed February 1, 2026, https://stackoverflow.com/questions/46085748/define-size-for-dev-shm-on-container-engine
42. Kubernetes Egress: Egress Network Policy, Istio Gateway, and More | Solo.io, accessed February 1, 2026, https://www.solo.io/topics/istio/kubernetes-egress
43. Network Policies - Kubernetes, accessed February 1, 2026, https://kubernetes.io/docs/concepts/services-networking/network-policies/
44. Securely deploying AI agents - Claude API Docs, accessed February 1, 2026, https://platform.claude.com/docs/en/agent-sdk/secure-deployment
45. Pod Security Standards - Kubernetes, accessed February 1, 2026, https://kubernetes.io/docs/concepts/security/pod-security-standards/
46. Pod Security - Amazon EKS - AWS Documentation, accessed February 1, 2026, https://docs.aws.amazon.com/eks/latest/best-practices/pod-security.html
47. OpenDevin: Code Less, Make More - GitHub, accessed February 1, 2026, https://github.com/AI-App/OpenDevin.OpenDevin