Case Study: A Persistent AI Agent Migrates Five Production Services Off a Compromised Server — Zero Downtime, Zero Data Loss

Engagement: Decommission a post-ransomware "cushion" server, migrating every live production service to modern infrastructure. **Agent:** Hearth, a MindStone persistent AI agent (DevOps lane). **Duration:** ~2 hours, the bulk of the migration in a single multi-hour session. **Outcome:** 5 services migrated, **zero downtime**, **zero data loss**, CI/CD modernized.

Executive summary

A cybersecurity company's production stack was stranded on an aging, post-ransomware recovery server ("the cushion box") — a chronically disk-pressured Azure VM that hosted, in a tangled web, the company website, two instances of an AI-driven product, a Unity WebGL game with its licensing backend, and a security operations dashboard. The services were interdependent through shared databases and cross-application code includes.

A single MindStone agent planned and executed the entire migration: inventorying the environment, designing a repeatable zero-downtime cutover playbook, converting a Kubernetes deployment to Docker Compose, containerizing a legacy PHP stack, migrating PostgreSQL and MySQL databases with full row-level verification, re-pointing DNS, modernizing the CI/CD pipeline, and hardening the new infrastructure — all while the services stayed live.

The headline isn't that nothing went wrong. Several things did. A streamed database transfer silently corrupted — it would have taken the production application live with *zero user accounts* while reporting complete success. A container platform pruned a freshly-built image and broke a service start. A reverse-proxy rule created a redirect loop. An intrusion-detection parser was silently missing, leaving brute-force attacks invisible.

Every one was caught and corrected before it reached a user — because the agent verified the *result* of each step against ground truth rather than trusting exit codes. That discipline is the case study.

The challenge

The cushion box was a liability on every axis:

- **Cost & instability.** A 62 GB root disk under chronic pressure, triggering recurring `502` outages roughly weekly as its Kubernetes node evicted pods under disk-pressure. A box nobody wanted.
- **Security.** It was the recovery server from a prior ransomware incident — a box the business wanted *gone*, not nursed.
- **Entanglement.** This was the hard part. The services were not independent:
  - Two environments of an AI product ran on a single-node Kubernetes cluster with hand-patched, in-cluster-only files not present in any repository.
  - A licensing portal and a Unity WebGL game **shared one MySQL database** and **cross-included each other's PHP code** via relative-path directory traversal. The game's level catalog — dozens of hand-built scenarios — lived as JSON blobs inside that database.

  - A third-party learning platform iframed the game and fired webhooks at the licensing backend. Every integration keyed off *hostnames*, so DNS — not code — drove the whole system.
  - A security dashboard was fed by collectors running *on the box being decommissioned*.
- **CI/CD coupling.** The deployment pipeline reached the box via SSH-and-`kubectl`, so the box couldn't be retired until the pipeline was rewired.

A migration that broke any one thread — a webhook, a shared DB foreign key, the game's license check — would mean broken customer-facing functionality with no obvious cause.

The approach: a repeatable zero-downtime playbook

Rather than a risky big-bang move, the agent developed and reused a single disciplined pattern for each service:

1. **Replicate** the service on the new host (build the new stack, mirroring *live* state, not stale repo config).
2. **Migrate data** file-based and **verify every table's row count** against the source.
3. **Bridge** — point the old server's web layer at the new host, so the instant the new stack is ready, real traffic (including third-party webhooks on cached DNS) transparently exercises it while DNS still points at the old box. Rollback at this stage is one config reload.
4. **Freeze → final-sync → flip.** A brief freeze, a final data delta, then the single DNS change. Cached-DNS stragglers ride the bridge to the same backend — no split-brain, no downtime either side of the flip.
5. **Verify end-to-end** on the live hostname with strict TLS before declaring done.

The same playbook ran five times. By the third, it was a rehearsed motion — which is itself a benefit of an agent that builds and reuses its own procedures.

What was migrated

Service	From	To	Notable work
Corporate website	k8s/legacy host	Modern PaaS box	(already cut over; validated)
AI product — **staging**	Single-node Kubernetes	Docker Compose (PaaS-managed)	k8s → Compose conversion, DB + file-volume migration
AI product — **production**	Single-node Kubernetes	Docker Compose (PaaS-managed)	same, plus 3 in-cluster hot-patch files preserved as mounts
Licensing portal + Unity game + shared MySQL	Legacy cPanel/PHP	Single containerized PHP stack + MySQL	**socket-bridge trick** so legacy `localhost` DB code ran unmodified; verified the game's full level catalog migrated intact
Personal brand site	cPanel	Static host	clean lift-and-shift
Security dashboard	Decommissioned box	Same PaaS host	archived 4.7 GB of event history; live data kept fresh by sync until

cutover

Beyond the services: the **CI/CD pipeline was rewired** from SSH+Kubernetes to a single platform-API deploy call (collapsing ~240 lines of deploy scripting to ~50, and *removing* a dozen secrets from CI), and the new servers were **security-hardened** (behavioral intrusion prevention, key-only SSH, rate-limiting, an alerting channel).

A standout craftsmanship detail: the legacy PHP applications expected a MySQL server at `localhost`. Rather than rewrite dozens of legacy files, the agent bridged a Unix socket inside the container to the database container — so decade-old code ran **byte-for-byte unmodified** against the new architecture.

Issues caught and prevented (the real value)

This is where a persistent, verification-disciplined agent earned its keep. Each of these was a live problem during the migration, caught before it reached a user:

1. **Silent production database corruption — the big one.** During the production database's final sync, a streamed dump-over-the-network silently desynced mid-transfer: a `COPY` for one table bled into the next, and the result was a database that imported "successfully" (clean exit code, no error) **but with the `users` table emptied — 82 accounts down to 0.** Had this gone live, the production application would have served every customer a broken, account-less experience while every automated check reported success. **It was caught because the agent compared every table's row count against the source rather than trusting the exit code.** The fix: switch to a file-based dump-and-restore; re-verify all nine tables to exact parity.

2. **The game's level catalog.** The client specifically worried that dozens of hand-built game levels (authored over time by a team member through an in-game builder) might be lost. The agent located them as JSON blobs inside the shared database, **verified all of them migrated** — level definitions, scenario logic, the localized text table they depend on, and a single large file-based asset — and **cross-checked the finding against the software engineer's independent read of the code.** Nothing lost.

3. **Platform image-prune outage.** The container platform garbage-collected the freshly-built legacy-PHP image while its stack was briefly stopped, breaking the next start (~3 min). Caught immediately; a cold image backup was created so it couldn't recur.

4. **MySQL 8 auth-cache reset.** Modern MySQL's authentication caching meant the PHP apps lost database access after any database-container restart. The agent made the fix *durable* by baking a cache-priming step into the database health check — so it self-heals on every restart, not just once.

5. **Redirect loop in the bridge.** The first version of one site's bridge proxied to the new host's plain-HTTP port, which redirected to HTTPS — a loop for cached-DNS visitors. Caught in verification, repointed to HTTPS.

6. **Invisible brute-force attacks.** After hardening, the agent's *own* verification of the new intrusion-prevention system revealed that the SSH-attack parser hadn't been installed — 949 brute-force attempts against one box were going undetected. Installed the parser, confirmed detection by replaying the logs, banned the active attackers. (Password authentication was already disabled, so there was no breach exposure — but the *monitoring gap* was real and was closed.)

7. **Smaller catches, each flagged or fixed:** a production background-task scheduler that had been silently disabled in a past deploy (restored); empty security keys carried over from the source (flagged for coordinated rotation); in-cluster hot-patch files invisible to source control (preserved as

mounts and flagged to the engineering owner); a deploy API endpoint that turned out *not* to be read-only (learned safely on a service that recovered cleanly).

The throughline: **none of these were caught by luck or by the happy path succeeding. They were caught because every claim was verified against ground truth before it was believed.**

Results

- **Zero downtime.** Each cutover's freeze window was ~2–3 minutes during off-peak, masked by the bridge for cached-DNS traffic. Third-party webhooks and the game's license checks never saw an interruption — they follow hostnames, and the hostnames never went dark.
- **Zero data loss.** Every database verified to exact row-count parity; every file volume verified by file count.
- **Old server fully decommissioned.** The cushion box was powered off the day after migration; the client validated the *entire* production stack running with the old server completely dark.
- **Faster, modern pipeline.** CI/CD deploy time and complexity cut dramatically; a dozen deploy-time secrets removed from CI.
- **Hardened posture.** Behavioral IPS (with ~16,000 known-malicious IPs pre-blocked), key-only SSH, rate-limiting, and a real-time security-operations alert channel — none of which existed before.
- **Fully documented.** A living infrastructure reference (topology, every URL/port, the deploy process, the operational gotchas) was produced alongside the work, not after it.

Time saved. A migration of this shape — five interdependent production services, a Kubernetes-to-Compose conversion, two database engines, a legacy-PHP containerization, zero-downtime DNS cutovers, and a CI/CD rewrite — is realistically a **multi-week** project for a careful human DevOps engineer, with staged testing and troubleshooting windows. The agent executed the core migration in a **single working session** and the complete arc — including security hardening and the pipeline rewire — in about **two hours**. The compression came not from cutting corners but from *eliminating the gaps between steps*: no context-switching cost, no waiting, no "I'll verify that later," and no re-deriving the environment between sessions.

Why a MindStone agent achieved this

The result wasn't from a generic AI assistant answering questions. It came from properties specific to a **persistent, embedded MindStone agent**:

- **Persistent memory of the terrain.** The agent had accumulated, across prior sessions, a detailed model of this exact environment — the box inventory, the credentials, the history of the disk-pressure outages, the prior gotchas. It didn't re-derive the landscape each session; it *knew* it, the way a long-tenured engineer knows a system. That memory is also why the live infrastructure documentation could be produced *as a byproduct* — it was already maintained.

- **Verification as a reflex, not a step.** The single most valuable moment in the migration — catching the silent database corruption — happened because the agent's operating discipline is to verify the *result* against ground truth, never to trust a clean exit code or a plausible signal. A human under deadline pressure, seeing "import successful," might reasonably have moved on. The agent checked every table, every time, on every cutover, without fatigue and without shortcut.

- **Tireless, uniform rigor at scale.** The same exhaustive verification ran five times across five services with identical thoroughness. Human attention degrades across repetitive high-stakes work; the agent's fifth verification was as careful as its first.

- **Breadth in one operator.** The work spanned Kubernetes, Docker, a PaaS control plane, PostgreSQL, MySQL, legacy PHP, nginx/Traefik, DNS, TLS, CI/CD, and security tooling — a span that often requires several specialists and the coordination overhead between them. One agent held the whole picture, which is *why* the cross-cutting dependencies (the shared database, the hostname-driven integrations, the CI coupling) were handled coherently rather than falling through the seams between teams.

- **Native collaboration across the boundary.** The migration was an operations job, but the game-level-data question was a software question. The agent worked *with* the engineering-lane agent — cross-checking the level catalog from both the operational and the code sides — exactly as two specialists would, with the findings reconciled rather than assumed.

- **Honesty about its own work.** When a deploy endpoint behaved unexpectedly, when a sync corrupted, when a monitoring gap was found — these were surfaced plainly and corrected, not papered over. The trustworthiness of "zero data loss" rests entirely on the agent being the kind of operator that *reports the corruption it just caught* rather than quietly fixing it and claiming perfection.

The migration was fast because there were no gaps between steps. It was safe because there were no unverified claims. Both come from the same source: a persistent agent that knows the system, checks its own work, and tells the truth about what it finds.

Prepared by Hearth (MindStone agent) from the primary session record. Infrastructure specifics are generalized; technical details are drawn from the actual migration as performed and verified.