Governing the Tools, Not the Agent: A Capability-Governed Deployment Platform for AI-Assisted Operations

Luis Dehlwes[a,*] Claude Code[b]

[a] *Legacy AI Labs* [b] *Anthropic*

∗ Corresponding author: contact@dehlwes.net

Abstract

Self-hosting the deployment of a small fleet of services forces an awkward choice. Full orchestrators such as Kubernetes and Nomad impose operational complexity out of all proportion to a one-person estate, whereas lightweight self-hosted platform-as-a-service (PaaS) tools such as Dokku, CapRover and Coolify streamline the human workflow but were never designed for a second, non-human operator, namely a large-language-model (LLM) agent, to drive them safely. We present **Nexus**, a self-hostable deployment platform built on a single thesis: expose every operation, to humans and to AI agents alike, through **one audited surface of capabilities**, each tagged with a **risk class**, so that an agent may observe and propose freely yet can never unilaterally perform an irreversible or outward-facing action without a **human-issued, per-resource grant**. We describe the design methodology (deliberate scope reduction, a convention-based deploy contract, capability-based least-privilege governance, a single human/agent surface with asymmetric authorization, and coordination-light high availability over a shared database) and its implementation as a governed engine over Docker and Caddy, exposed to agents through the Model Context Protocol. We position Nexus against existing PaaS and agent-tooling work and report characteristics of the running system, which today governs a real three-machine fleet.

Index Terms: platform-as-a-service, LLM agents, capability-based security, least privilege, continuous delivery, self-hosting, Model Context Protocol.

I. Introduction

The operator of a handful of web services on their own hardware is poorly served by today's deployment tooling. At one extreme, container orchestrators such as Kubernetes [1] (descended from Google's Borg [2]) and HashiCorp Nomad [3] offer powerful scheduling and self-healing, but their control-plane, networking and configuration surface imposes a learning and maintenance cost that a single operator rarely recoups; operational complexity and the attendant skills gap remain among the most-cited barriers to cloud-native adoption [4]. At the other, a generation of lightweight self-hosted PaaS tools, namely Dokku [5], CapRover [6] and Coolify [7], reproduce the ergonomics that Heroku [8] popularized: push code, get a running, TLS-fronted service. These tools succeed by *reducing scope*, adopting the conventions of the twelve-factor app [11] and continuous delivery [12].

A new actor now wants to use the same tooling. LLM agents can decompose goals and invoke external tools [15], [16], a growing body of work studies them as autonomous operators [17], [18], and they have begun to drive real operations workflows [20]. The Model Context Protocol (MCP) [19] standardizes how such an agent discovers and calls the tools of an external service. It is therefore increasingly tempting to let an agent *operate infrastructure*: create apps, set secrets, cut over releases. Yet none of the self-hosted platforms above were built for a caller that may act without human intent behind each request, and controlled studies find that tool-using agents commit high-stakes errors at non-trivial rates [21]. Handing an autonomous process the same unconditional deploy-and-secrets access a human enjoys is a standing invitation to irreversible mistakes and to prompt-injection-driven misuse.

We argue that the right place to resolve this tension is the *deploy boundary* itself, and that the correct tool is decades old: capability-based access control and the principle of least privilege [22]. Our thesis is that a deployment platform should present *one* surface of explicitly enumerated, individually risk-classed capabilities to humans and agents alike, and that authorization should be *asymmetric*: an agent may run read-only and reversible operations, but every irreversible or outward-facing action requires a grant issued by a *different* principal: a human. The agent looks and proposes; the human authorizes what cannot be undone.

This paper makes three contributions. (1) We articulate a design **methodology** for agent-native but governed operations, resting on scope reduction, a convention-based deploy contract, capability governance, a single human/agent surface, and coordination-light availability (Section II). (2) We describe the **implementation** of Nexus, a working system realizing that methodology over Docker and Caddy (Sections III–IV). (3) We **position** Nexus against PaaS and agent-tooling work and reflect honestly on where it helps and where it does not (Sections V–VI). Nexus is an implemented system, not a controlled study; we report characteristics of the running

deployment rather than benchmarks.

II. Methodology and Design Principles

Nexus was not derived from a scheduling problem but from a governance one. Its design follows five principles; the first two situate it among conventional PaaS tooling, the middle two carry the paper's contribution, and the last keeps availability cheap.

A. M1: Deliberate scope reduction

The estate we target is a single operator with a small, long-lived set of services on one to a few machines. From this we deliberately implement only the fraction of a PaaS such an operator actually exercises: source-to-URL deploys, domains with automatic TLS, secrets, service-to-service wiring, and a handful of backing stores, and omit autoscaling, multi-tenant scheduling and a bespoke overlay network. Scope reduction is not a limitation to be apologized for but the enabling move: it is what makes the remaining surface small enough to enumerate, risk-class and audit in full (Section II-C).

B. M2: A convention-based deploy contract

Rather than per-app configuration, Nexus fixes a small contract in the spirit of the twelve-factor app [11] and continuous delivery [12]. A conformant service listens on an injected `$PORT`, exposes a readiness signal, and receives its configuration and a managed inbound token through the environment. State lives declaratively in a relational database, and the running infrastructure (reverse-proxy routes, mesh aliases, container set) is brought toward that declared state by idempotent, *level-triggered* reconciliation rather than one-shot imperative steps, the control discipline that also underpins GitOps [13] and infrastructure as code [14]. A diff gate ensures an unchanged declaration never re-applies.

C. M3: Capability-based, risk-classed governance

Every operation the platform can perform is modeled as a named *capability* with a machine-readable input schema and one of three *risk classes*, directly echoing the least-privilege discipline of Saltzer and Schroeder [22] and the capability model of Dennis and Van Horn [23], [24]:

- **read**: observe state; runs immediately.
- **update**: change persisted configuration; reversible; may require an explicit confirmation.
- **external**: build and cut over real releases, or delete; irreversible or outward-facing.

A single *executor* is the only path to side effects: it validates input against the schema, checks the caller's authorization for the capability's risk class, records an audit entry for *every* attempt (including refusals), providing the visibility that governing autonomous agents demands [26], and only then dispatches. Because the surface is closed and enumerated, an operation that is not a capability simply cannot be performed: the object-capability property that authority is never ambient [24].

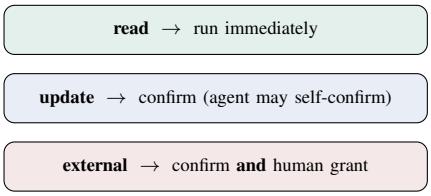

Fig. 1. Asymmetric authorization by risk class. An agent may self-authorize read and update actions but can never satisfy the grant requirement of an external action; that grant must be issued by a different, human principal.

D. M4: One surface, asymmetric authorization

Humans (through a REST/web interface) and agents (through MCP [19]) invoke the *same* executor over the *same* capabilities; there is no privileged back channel and no separate "AI API" to drift out of sync. What differs is authorization, and it is deliberately asymmetric (Fig. 1). A **read** runs for anyone. An **update** runs with an explicit confirmation, which an agent may supply for itself. An **external** action, however, requires a *grant* pre-issued by a *different* principal: an agent's own confirmation is refused, and a human operator must have authorized that specific app. Thus an agent can design an entire system (create apps, wire them, stage secrets as fillable placeholders) and still be structurally unable to push an irreversible release or delete data on its own authority. Human-in-the-loop is not a policy bolted on top but a property of the authorization algebra; the grant is a scoped, auditable, human-issued delegation of authority, echoing recent proposals for authenticated agent authorization [27].

E. M5: Coordination-light high availability

Rather than a consensus-backed control plane, Nexus peers share a single relational database and coordinate through it. Each in-flight deployment is *owned* by the peer driving it, proven alive by a heartbeat; a reaper reclaims only work whose driver has gone silent, and each peer reconciles its own reverse proxy from the shared truth. This trades the generality of an orchestrator for radically less moving machinery (no scheduler, no overlay, no quorum), which is the right trade at single-operator scale.

F. Engineering discipline

The implementation is convention-over-configuration and verification-driven: the current system carries 282 automated backend tests that gate every change, and behavioral changes are exercised against a running instance before release. This is process, not proof; we claim tested software, not a verified system.

III. Implementation and Architecture

Nexus is a TypeScript monorepo comprising a governed *engine*, a per-machine *agent* that speaks to the local Docker daemon, a Caddy reverse proxy, and a shared PostgreSQL database; agents reach it via an MCP server and humans via a

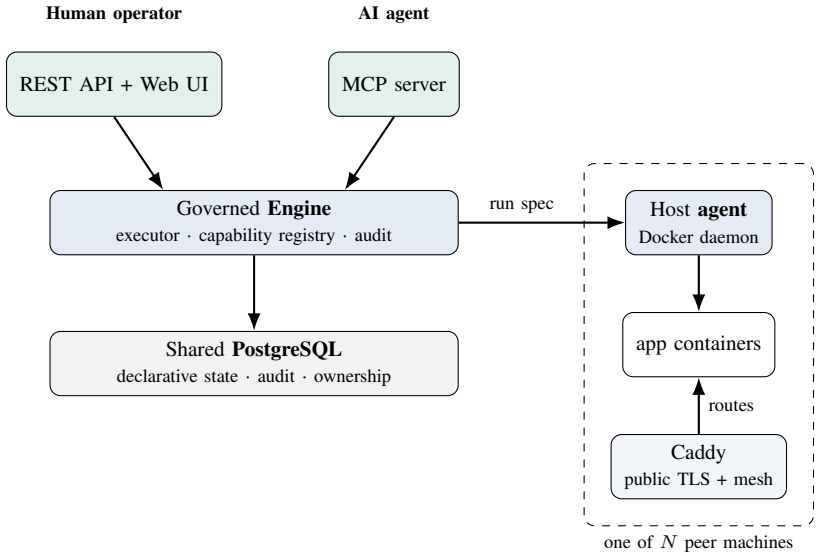

Fig. 2. Nexus architecture. Humans and agents invoke the *same* governed engine over the same capabilities; the engine writes declarative state to a shared database that every peer machine reconciles its own Docker and Caddy against. The public reverse proxy and an internal service mesh are the same Caddy instance.

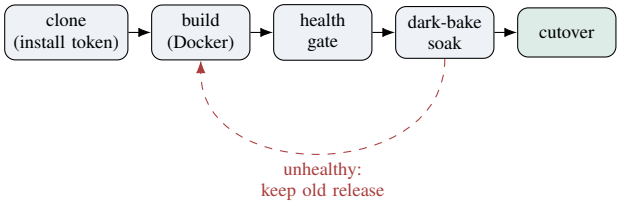

Fig. 3. Stateless zero-downtime deploy: the live release is replaced only after a new one proves healthy through a soak window; otherwise it is discarded and the old release stands.

REST API and web UI (Fig. 2). It stands on Docker and Caddy and implements only the deployment logic between them.

A. The capability layer

The governed surface currently comprises **46** capabilities: 15 *read*, 28 *update* and 3 *external*, of which 27 require explicit confirmation. The three external capabilities (deploy a release, deploy a whole group, and delete an app) are exactly the irreversible, outward-facing ones, and each is gated by a per-resource grant; group deploys expand the check to every member so that an agent cannot launder an ungranted app's release through a group it belongs to. Every capability description doubles as the machine-readable specification the agent reads, so the human documentation and the agent's contract cannot diverge.

B. The deployment pipeline

A deploy (Fig. 3) clones the app's source using a short-lived GitHub App *installation token*, builds a container image, and then, for a stateless web service, runs the full zero-downtime pipeline: a health gate, a *dark-bake* soak during which the new release must stay healthy, and a transactional cutover of the reverse-proxy route. A build that never becomes healthy never replaces the live one. Stateful services and stores instead stop the old container before starting the new, accepting a brief downtime with automatic rollback if the replacement fails to go live. An optional release command (e.g. a database migration) runs once before the app container and fails the deploy without promoting on a non-zero exit.

C. Peer cluster and reconciliation

Machines join by exchanging a pairing code and thereafter share the database. A claim loop lets any peer pick up queued work for apps pinned to it; per-deploy ownership and a heartbeat let a reaper distinguish a crashed driver from a living one and recover only the former. Each peer independently reconciles its Caddy configuration from the shared state, diffing to avoid needless reloads, so the public and internal routing converge without any central router.

D. Service mesh and credential wiring

Linked services reach each other over an internal, in-network Caddy mesh; the consumer receives injected `<PROVIDER>_URL` and `<PROVIDER>_TOKEN` variables on its next deploy and never hard-codes an address. Shared-credential *groups* inject a common environment into every member and, per a chosen topology (full mesh, hub-and-spoke star, or none), auto-wire members to one another. Because wiring variables are nexus-managed, a user variable can never silently shadow a real credential.

E. Secrets and supply chain

Secret values are encrypted at rest under a master key and are redacted from logs and reads; an agent may *order* a server-minted secret or stage a named placeholder without the value ever passing through its context. Source access uses a GitHub

TABLE I
QUALITATIVE POSITIONING OF NEXUS AMONG SELF-HOSTING AND ORCHESTRATION OPTIONS (✓ = YES, ∼ = PARTIAL OR VIA ADD-ON, × = NO, – = NOT APPLICABLE).

System	Self-hosted	Light-weight	Auto-TLS	Multi-machine	Agent-native	Primary niche
Heroku [8]	×	–	✓	✓	×	managed PaaS, hobby to business
Fly.io [9], Render [10]	×	–	✓	✓	×	managed edge/app hosting
Dokku [5]	✓	✓	✓	×	×	single-host push-to-deploy
CapRover [6]	✓	✓	✓	∼	×	self-hosted PaaS with a UI
Coolify [7]	✓	∼	✓	∼	×	self-hosted multi-service PaaS
Kubernetes [1]	✓	×	∼	✓	×	cluster orchestration at scale
Nomad [3]	✓	∼	∼	✓	×	flexible workload scheduler
Nexus (this work)	✓	✓	✓	✓	✓	single-operator, agent-operated fleet

App with read-only Contents permission and outbound polling rather than inbound webhooks, so the platform functions on a private, tailnet-only host with no public ingress.

IV. WHAT NEXUS DOES

For the operator, Nexus turns a GitHub repository or a prebuilt image into a running, TLS-fronted service reached at a per-app domain; lets them set environment and secrets, wire services together, attach recipe-driven backing stores, expose or retract public domains, span several machines as one mesh, and inspect per-machine disk usage, and it can update itself. For the agent, the *same* operations are available as MCP capabilities under the governance of Section II-C: it can inspect the fleet, design and stage a multi-service system, and propose deployments, while every build-and-cutover waits on a human grant. The two operators share one audited history.

V. RELATED WORK AND COMPARISON

Self-hosted PaaS and orchestrators. Heroku [8] defined the push-to-deploy experience that Dokku [5], CapRover [6] and Coolify [7] bring to self-hosting, while managed platforms such as Fly.io [9] and Render [10] offer it as a service. Kubernetes [1] and Nomad [3] generalize to cluster scheduling at the cost of operational weight. Table I situates Nexus: it shares the light, self-hostable, auto-TLS character of the Dokku/Coolify family, but is distinguished by a first-class, risk-classed *governance* surface shared by humans and AI agents, a dimension none of the others target.

Agents and tool use. Techniques such as ReAct [15] and Toolformer [16] let LLMs invoke external tools, surveys chart their use as autonomous agents [17], and MCP [19] standardizes the tool interface Nexus exposes. This literature concerns how an agent *calls* tools; a newer strand studies the *risks* of tool-using agents [21], visibility and logging for deployed agents [26], and authenticated delegation of scoped authority to them [27]. Nexus is a concrete, running instantiation of that agenda at the deploy boundary (risk-classed capabilities, per-action human grants, and a complete audit of every attempt), and thus governs the tools rather than improving the caller.

Security model. The governance rests on established foundations, namely least privilege and the failure of ambient authority [22], and the capability model [23]–[25] , applied to a contemporary problem: bounding the blast radius of an autonomous, possibly-manipulated agent at the deploy boundary.

VI. DISCUSSION

Where it helps. A single governed surface removes the drift between what a human can do and what an automation can do, and makes least privilege the default rather than an afterthought: the worst an ungranted agent can do is propose. The coordination-light peer model gives multi-machine resilience without an orchestrator to operate. The natural application domains are solo developers and small teams, homelab and edge fleets, and, increasingly, estates an operator wants an AI agent to help run.

Characteristics of the running system. Nexus is not a paper design. The reference deployment governs a three-machine fleet (a database/ingress host and two build peers) and currently serves 29 live app releases; the governed surface is the 46 capabilities analyzed above; the control-plane web bundle is code-split so that pages ship without the graph-editing dependency (a 24% reduction of the initial bundle). These are properties of the implementation, offered as evidence that the design is buildable and operable, not as a controlled evaluation.

Limitations. The design targets single-operator, small-fleet scale and makes no claim beyond it; against a cluster orchestrator at scale, Kubernetes and Nomad remain the right tools. The security model is argued, not formally verified or externally audited, and the asymmetric-authorization property rests on the correctness of the executor and grant store. Our comparison is qualitative; we report no controlled performance study. Finally, a deployment reachable only over a private network today serves its UI over plain HTTP, which disables browser secure-context APIs, a pragmatic gap closed by fronting it with TLS.

VII. CONCLUSION

Nexus takes the position that as autonomous agents begin to operate real infrastructure, the deployment platform, not the

agent, is where safety should be enforced, and that the enforcement mechanism is the well-understood pairing of capability-based access control with least privilege. By presenting one enumerated, risk-classed surface to humans and agents alike and making authorization asymmetric, a small platform can let an agent design and propose freely while structurally reserving every irreversible act for a human. The result is a buildable system, running a real fleet, that is agent-native without being agent-trusting. Future work includes automatic TLS via DNS-01 to close the secure-context gap, a formal treatment of the grant algebra, and a broader evaluation across operators.

REFERENCES

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, 2016. [Online]. Available: https://queue.acm.org/detail.cfm?id=2898444

[2] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. EuroSys*, 2015.

[3] HashiCorp, "Nomad Documentation," accessed Jul. 2026. [Online]. Available: https://developer.hashicorp.com/nomad

[4] Cloud Native Computing Foundation, "CNCF Annual Survey 2024," 2025. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2024/

[5] Dokku, "The smallest PaaS implementation you've ever seen," accessed Jul. 2026. [Online]. Available: https://dokku.com

[6] CapRover, "Scalable, free and self-hosted PaaS," accessed Jul. 2026. [Online]. Available: https://caprover.com

[7] Coolify, "An open-source & self-hostable Heroku/Netlify alternative," accessed Jul. 2026. [Online]. Available: https://coolify.io

[8] Heroku, "How Heroku Works," Heroku Dev Center, accessed Jul. 2026. [Online]. Available: https://devcenter.heroku.com/articles/how-heroku-works

[9] Fly.io, "Developer Documentation," accessed Jul. 2026. [Online]. Available: https://fly.io/docs

[10] Render, "Documentation," accessed Jul. 2026. [Online]. Available: https://render.com/docs

[11] A. Wiggins, "The Twelve-Factor App," 2011. [Online]. Available: https://12factor.net

[12] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[13] A. Richardson and W. Denniss, "GitOps: Operations by Pull Request," *KubeCon + CloudNativeCon North America*, 2017.

[14] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media, 2016.

[15] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," in *Proc. ICLR*, 2023. arXiv:2210.03629.

[16] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language Models Can Teach Themselves to Use Tools," in *Proc. NeurIPS*, 2023. arXiv:2302.04761.

[17] L. Wang, C. Ma, X. Feng *et al.*, "A Survey on Large Language Model based Autonomous Agents," *Frontiers of Computer Science*, vol. 18, art. 186345, 2024.

[18] Z. Xi, W. Chen, X. Guo *et al.*, "The Rise and Potential of Large Language Model Based Agents: A Survey," arXiv:2309.07864, 2023.

[19] Anthropic, "Introducing the Model Context Protocol," Nov. 2024. [Online]. Available: https://www.anthropic.com/news/model-context-protocol

[20] L. Zhang, T. Jia, M. Jia *et al.*, "A Survey of AIOps in the Era of Large Language Models," arXiv:2507.12472, 2025.

[21] Y. Ruan, H. Dong, A. Wang, S. Pitis, Y. Zhou, J. Ba, Y. Dubois, C. J. Maddison, and T. Hashimoto, "Identifying the Risks of LM Agents with an LM-Emulated Sandbox," in *Proc. ICLR*, 2024. arXiv:2309.15817.

[22] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[23] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Commun. ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[24] M. S. Miller, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," Ph.D. dissertation, Johns Hopkins Univ., 2006.

[25] M. S. Miller, K.-P. Yee, and J. S. Shapiro, "Capability Myths Demolished," Johns Hopkins Univ. Systems Research Lab., Tech. Rep. SRL2003-02, 2003.

[26] A. Chan, C. Ezell, M. Kaufmann, K. Wei *et al.*, "Visibility into AI Agents," in *Proc. ACM FAccT*, 2024. arXiv:2401.13138.

[27] T. South, S. Marro, T. Hardjono, R. Mahari, C. D. Whitney, D. Greenwood, A. Chan, and A. Pentland, "Authenticated Delegation and Authorized AI Agents," arXiv:2501.09674, 2025.