TECHNICAL WHITE PAPER: **July 2026**

Running Claude Agents on VMware Tanzu Platform

A reference architecture for hosting the Claude Agent SDK on a private, governed platform

vmware® by Broadcom

A technical white paper for Tanzu Platform customers

Featuring VMware Tanzu Platform 10.4 and the Claude Agent SDK, June 2026

- **Audience:** Platform engineers and architects
- **Stakeholders addressed:** Platform Engineering, Security and Compliance, AI Engineering, CIO/CTO
- **Companion resource:** the tanzu-claude-agent proof-of-concept repository and live deployment.

About this paper

This paper presents a reference architecture for running agents built on the Claude Agent SDK as first-class applications on VMware Tanzu Platform: the agent loop and enterprise harness on the foundation, the services around the agent managed by the platform, and model inference as the only external dependency.

A working proof of concept grounds every section. Architecture, security model, data and memory design, day-2 operations, and a deployment walkthrough are covered in turn.

Table of contents

1. Executive Summary

Enterprises have spent two years proving that large language models are useful and one year discovering that the hard part is everything around the model. An agent that can read repositories, query infrastructure, file tickets, and remember what it learned needs credentials, network reach, storage, identity, and audit. Those are platform concerns, and most AI offerings answer them with someone else's platform.

This paper describes a different arrangement, and a working implementation of it. The Claude Agent SDK packages the agent runtime that powers Claude Code (the tool loop, session management, permissions, hooks, and skills) as a library any application can embed. Deploy that application on VMware Tanzu Platform and the division of labor becomes unusually clean: Anthropic provides model inference through a single TLS connection, and your foundation provides everything else. The agent loop runs in a buildpack-built container you patch. Its tools reach internal systems through MCP servers hosted on the platform and fronted by the MCP Gateway, which authenticates every call against your SSO and writes every tool invocation to an audit log. Its memory and vector store live in a bound Tanzu for Postgres instance. Its skills and working files sit on an NFS volume shared across instances. Its secrets never leave CredHub and service bindings.

The architecture is demonstrated by a proof of concept, tanzu-claude-agent: a Node.js application of modest size, deployed and reachable on a Tanzu foundation today. It streams agent activity to a browser in real time and deploys with cf push. The services it binds (source control, platform operations, memory, mail) reflect what its demos need; a production deployment binds whatever its own tools require, and changing that set is a manifest edit, not an architecture change. Nothing in the POC required a new operational discipline; the platform team runs it the way they run every other app, which is precisely the argument.

The intended reader is a platform engineer or architect deciding how their organization will run agents in production. The paper presents the architecture, its security model, the data and memory design, and day-2 operations, then closes with a deployment walkthrough.

2. The Problem: Agents Without a Platform

Teams that set out to build an agent on the raw Messages API discover quickly that the model is the easy dependency. The API returns one model response per call. The loop around it (deciding which tool the model asked for, executing it, feeding results back, compacting context as the conversation grows, retrying transient failures, persisting the session) is application code someone has to write and maintain. Most teams write it badly once, then write it again. The framework ecosystem exists because this loop is genuinely hard to get right, but adopting a framework trades one maintenance problem for another and still leaves hosting unanswered.

Hosted agent products answer the loop question and create a governance one. When the agent executes on vendor infrastructure, every integration with an internal system means opening a path from someone else's cloud into yours. Binding the agent to a database that holds regulated data means that data transiting infrastructure your security team cannot inspect. For a bank or an insurer, each such connection is a negotiation with risk and compliance; ten tools means ten negotiations. The pattern that makes hosted agents convenient for demos makes them expensive for production in regulated environments.

The third path, and the most common one in practice, is the laptop. A developer runs an agent script locally with a personal API key and personal GitHub token, and it works. It works right up until a second developer wants it, an auditor asks whose credentials the agent used last Tuesday, or the script needs to run while its author is on vacation. Personal tooling does not scale into shared infrastructure, and credentials that live in dotfiles do not survive a security review.

What is missing from all three is not intelligence but operational substrate: a place where the agent runs under the same identity, network, secrets, and observability regime as the rest of the production estate. Enterprises already own such a place.

3. The Approach: The Agent Loop On-Platform

The Claude Agent SDK changes the build-versus-host calculation because it extracts the agent runtime from Claude Code and ships it as an embeddable library. The query() call gives an application the production loop Anthropic runs at scale: autonomous tool use until task completion, file and search tools included, multi-turn sessions with resume, a permission system with hooks for enterprise guardrails, and two extension mechanisms (skills and MCP) that let behavior grow without code changes. The harness application around it can be small. In the proof of concept it is a few thousand lines, most of which serve the UI.

A library, unlike a hosted product, runs where you put it. Putting it on Tanzu Platform is not an arbitrary choice; it is the placement that makes the enterprise constraints from the previous section dissolve into existing platform features. Credentials become service bindings. Network policy becomes the foundation's deny-by-default posture with one explicit egress to api.anthropic.com. Scaling, routing, health management, log aggregation: all inherited, none built.

This is a deliberate contrast with Claude Managed Agents, Anthropic's hosted execution environment, and the distinction matters enough to state plainly. Managed Agents run the loop in Anthropic-operated sandboxes, which is the right trade for teams without a platform. Organizations that operate a Tanzu foundation already have somewhere better: a runtime they control, with the governance machinery regulators expect already in place. Hosting the SDK keeps the loop, the tools, the data, and the audit trail on your side of the wire, and sends Anthropic only what inference requires.

Broadcom's own direction confirms the pattern. Tanzu Platform 10.4 introduced agent foundations, a secure-by-default agentic runtime built on exactly the principles this architecture uses: buildpack-based supply chain instead of hand-rolled Dockerfiles, structural isolation of secrets between agents, zero-trust networking where connectivity exists only through explicit service bindings, and a centralized gateway for models and tools. The reference architecture in the next section is one concrete, currently deployable expression of that direction, built from generally available platform services and the Agent SDK.

4. Reference Architecture

The architecture described here is a pattern, not a product, and it is runnable today. A proof of concept, an open repository called tanzu-claude-agent, is deployed on a Tanzu Platform foundation and exercises every component behind a working chat interface. Throughout this section the POC supplies the concrete examples; treat its particular service bindings and tool choices as illustrations of the mechanism, not as a prescribed bill of materials. Your deployment will bind fewer services or more, and different ones.

The section builds the architecture the way you would deploy it. Figure 1 starts with the only piece that contains application code, the agent harness. Figure 2 introduces the mechanism that attaches everything else to it, the service binding. Figure 3 then assembles the full picture, and by that point every box in it should feel inevitable.

The design splits responsibility along one line. Anthropic's API provides model inference, and nothing else. Every other moving part of the agent (the loop that drives it, the tools it calls, the credentials it holds, the memory it accumulates, the files it reads and writes) runs inside the foundation as platform-managed components. A regulator asking "where does the agent live?" gets a one-sentence answer: in your data center, on your platform, behind your network policy.

4.1 The agent harness

The harness, shown in Figure 1, is a Node.js application that embeds the Claude Agent SDK and exposes it over HTTP. It is deliberately thin. The SDK's query() call carries the weight: it runs the same agent loop that powers Claude Code, with tool orchestration, session management, automatic retries, context compaction, and a permission system, none of which the application team had to build.

Figure 1. The agent harness: building blocks of a single CF app

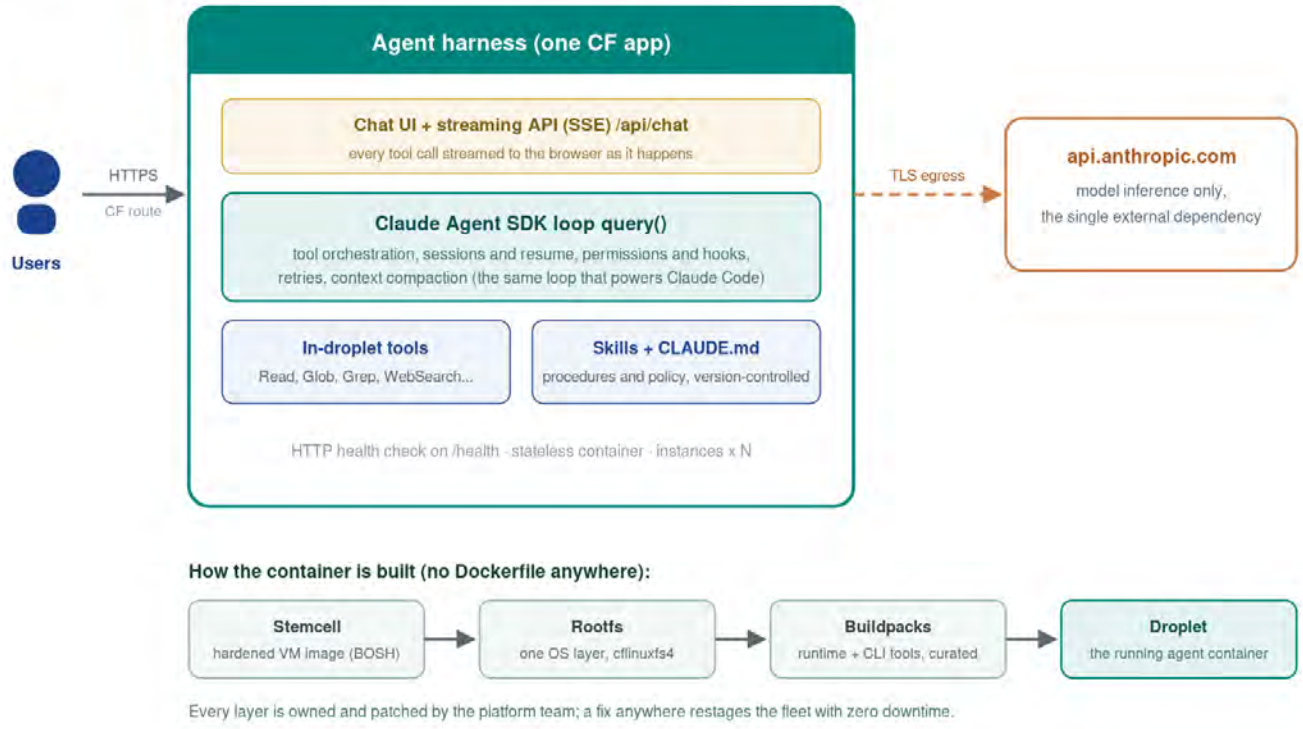

Figure 1. The agent harness: the building blocks of a single CF app

The app's own code concerns itself with serving a chat UI, streaming agent activity to the browser over SSE, and translating platform facts (bound services, SSO identity) into SDK configuration.

It deploys like any other CF app. The manifest declares buildpacks rather than referencing an image; in the POC, apt-buildpack chained before nodejs_buildpack puts the CLI tools its agent needs (`ripgrep`, `gh`) into the droplet without anyone writing a Dockerfile, and yours would declare whatever tools your agent calls. An HTTP health check on `/health`, a memory quota, `cf push`. This is the point of the whole exercise: the agent is an app, and the platform already knows how to run apps. The buildpack choice also carries a security argument substantial enough to get its own treatment in section 5.

Two details in the harness deserve attention because they carry the governance story. First, the tool surface is a policy decision, not an accident. The harness defines a base tool set (Read, Glob, Grep, WebSearch, WebFetch, Skill) and withholds Bash, Edit, and Write unless an operator enables shell access explicitly; the agent's behavioral contract lives in a version-controlled CLAUDE.md, and its task-specific competence in a `skills/` directory next to the code. Second, the harness resolves its MCP configuration from `VCAP_SERVICES` at startup. Adding a tool server to the agent means binding a service, not editing code.

4.2 Services by contract: the broker and the binding

Everything the rest of this section attaches to the harness arrives through one mechanism, and it is worth understanding before the boxes multiply, because it carries more of the security model than any firewall rule. Figure 2 traces it. A service broker is a component that advertises capabilities (a Postgres plan, an NFS volume, an SSO instance, an MCP server) in the platform marketplace and knows how to provision them on demand. An operator runs `cf create-service` and the broker builds a dedicated instance; the operator runs `cf bind-service` and the broker mints credentials for exactly that app and that instance. The credentials land in the platform's secret store, and at container start the platform injects them into the app's environment as `VCAP_SERVICES`, where the harness reads them and configures the SDK.

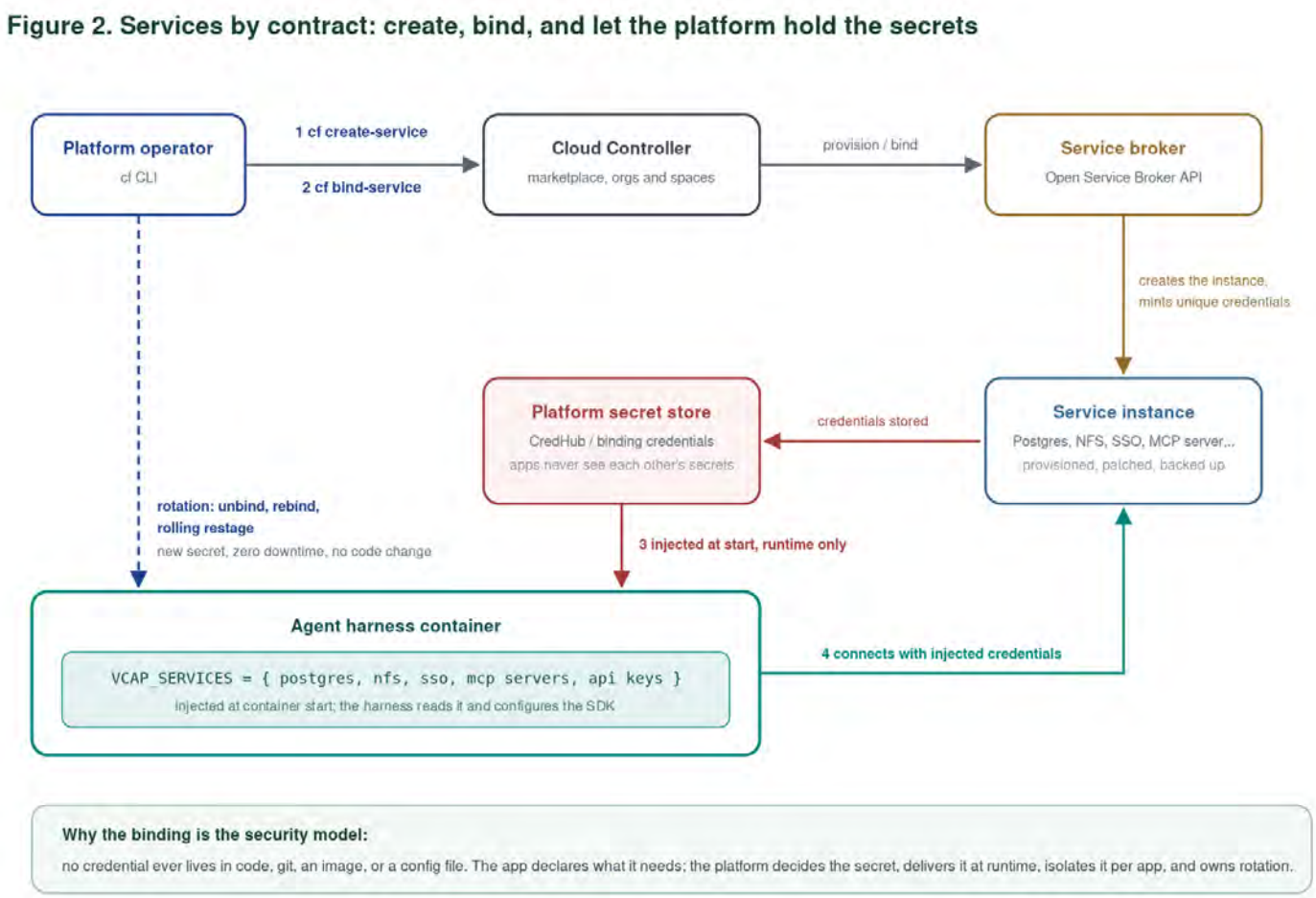

Figure 2. Services by contract: create, bind, and let the platform hold the secrets

Walk through what never happened in that sequence. No developer generated a password, saw one, or chose where to store it. Nothing was written to a config file, committed to git, or baked into an image, because there was never a moment when a human held the secret. The app declares what it needs; the platform decides the credential, delivers it at runtime only, and isolates it structurally so that no other app on the foundation can read it. Rotation, the chore that breaks quarterly in most estates, becomes an unbind, a rebind, and a rolling restage: new secret, zero downtime, no code change, owned entirely by the platform team. For an agent (a workload that will hold credentials to source control, databases, and operational systems simultaneously) abstracting credential management away from the application is not a convenience. It is the difference between an agent that can leak a secret and an agent that never possessed one.

4.3 MCP servers behind a gateway

The agent's reach beyond its own container comes from Model Context Protocol servers, and this is where most of the security posture concentrates. Each server arrives as a service binding; the harness reads `VCAP_SERVICES`, constructs the SDK's `mcpServers` configuration, and the agent discovers the tools at session start. The POC binds endpoints for source control (GitHub, GitLab), platform operations (CF and BOSH), mail, and memory, because those are what its demos exercise. A claims-processing agent would bind a document store and a policy system instead. The architecture does not care; the binding mechanism is the contract.

None of these servers is reached directly. They sit behind the MCP Gateway, a component of Tanzu AI Services 10.4 that gives the platform team one governed endpoint in front of every tool server. The gateway authenticates users against a configured identity provider—Tanzu SSO (OIDC), or a user-provided OIDC, OAuth, or GitHub Enterprise identity source—and issues per-user tokens; the harness stores each user's token in their session and injects it as a bearer header on every MCP connection that user's agent makes. The practical consequence: when the agent comments on a GitHub issue, it does so as the human who asked, not as a shared service account. The gateway also writes a structured audit log (the schema is published in the AI Services reference documentation), which turns "what did the agent touch?" from a forensic project into a query.

Off-platform APIs follow the same path. The gateway supports registering external MCP servers, so even tools that ultimately call github.com or an external SaaS present themselves to the agent through the same governed, audited front door. The agent itself needs exactly one egress permission: TLS to api.anthropic.com.

4.4 Postgres for memory and vectors

An agent without memory restarts life with every conversation. The pattern here is a bound Postgres instance with an MCP memory server in front of it (the POC uses one called tanzumem), giving the agent durable, multi-tenant fact storage: `save_fact`, `recall_facts`, scoped per user or conversation, with the agent's identity resolved server-side from the bound API key so the agent can never wander across tenant boundaries.

The same Postgres service carries the retrieval story. Tanzu for Postgres ships with pgvector, so embeddings of internal documents live in the operational database the platform team already backs up, patches, and monitors, rather than in a separate vector database with its own lifecycle. For most enterprise agent workloads, which retrieve from thousands to low millions of chunks, this is the right trade: one fewer system, one fewer credential, and recall quality that competes with dedicated stores at this scale. Teams that outgrow it can introduce a specialized store later without touching the harness, because retrieval is reached through MCP like every other tool.

4.5 A shared filesystem for skills and working files

Skills are the agent's institutional knowledge: versioned Markdown procedures (deploy checklists, audit runbooks, response-style rules) that the SDK loads on demand. Baking them into the droplet would work for a single instance, but it would mean a restage for every skill update and no shared scratch space once the agent scales out.

The deployment solves this with an NFS volume service. A `volume_mounts` entry in the manifest places a shared filesystem at a known path inside every container; the harness points `AGENT_WORKSPACE` and the skills directory at the mount, and all instances see the same files. Skill updates become a file copy instead of a restage. Agent working files survive container recycling. Two instances serving the same user session read the same workspace. The volume is an ordinary marketplace service, created with `cf create-service` and bound like the rest, so the shared filesystem inherits the platform's lifecycle instead of bringing its own.

4.6 The assembled picture

Figure 3 puts the pieces together, and the remaining boxes in it cost the application team nothing, which is the argument for building here rather than on bare infrastructure. Identity comes from a bound SSO instance; the Anthropic API key arrives through the same binding mechanism as everything else, via a user-provided service. Routes, TLS termination, horizontal scale, log aggregation, and metrics are platform defaults. Networking on Tanzu Platform 10.4 foundations is deny-by-default; the agent reaches api.anthropic.com because an operator allowed it, and nothing else because nobody did.

Figure 3. The assembled reference architecture: Claude Agent SDK harness on Tanzu Platform

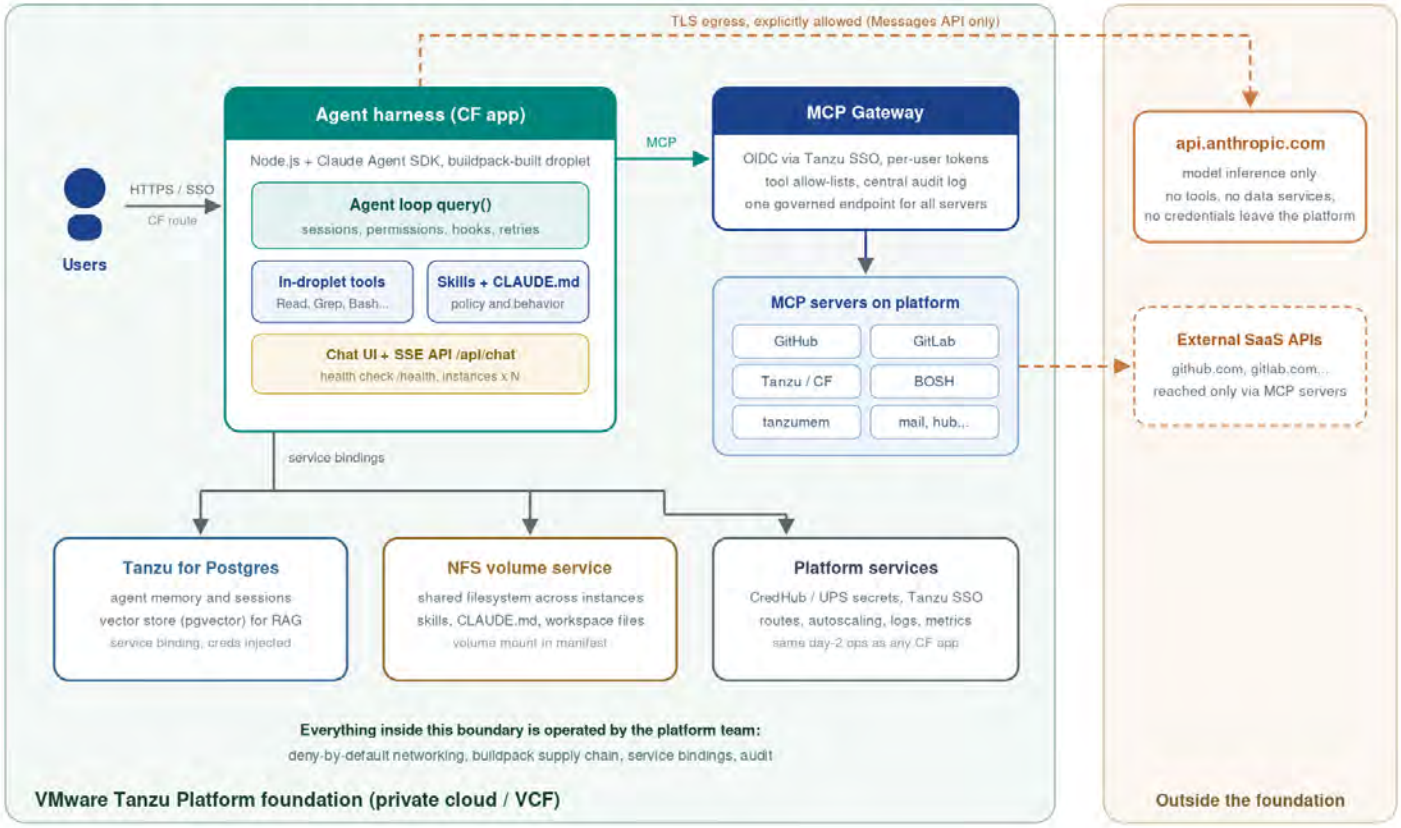

Figure 3. The assembled reference architecture on a Tanzu Platform foundation

A single request traces cleanly through the assembled architecture. A user authenticates through SSO and sends a message; the CF route delivers it to a harness instance; the SDK loop plans, calls tools in the droplet or through the gateway, consults memory in Postgres, and streams every intermediate step back to the browser. The only bytes leaving the foundation are the model exchange with Anthropic.

5. Security Model

Agent security differs from application security in one structural way: the code path is chosen at runtime by a model reasoning over inputs you do not fully control. A conventional app executes the branches its developers wrote. An agent decides, per request, which tools to call and with what arguments, and an attacker who can influence its inputs (a poisoned document in a retrieval corpus, a crafted issue comment, an email the agent was asked to summarize) is attempting to choose for it. The defensive consequence is that you cannot rely on the agent always behaving; you contain what it can do when it misbehaves. This architecture builds that containment in four layers: the supply chain that produces the container, the credentials regime, the network, and the tool surface itself.

5.1 Runtime containment and the supply chain

Start with the container, because everything else runs inside it. On Tanzu Platform the agent's image is not built by a developer from a Dockerfile; it is assembled by an opinionated buildpack pipeline that the platform team controls end to end. The stack is uniform from the bottom up: BOSH manages the stemcell (the hardened VM image every cell runs), the platform supplies a single rootfs (cflinuxfs4), and the buildpack layers the runtime and application on top. When a CVE lands in the OS layer, the platform team patches the stemcell or rootfs once and restages; every agent on the foundation picks up the fix without any application team rebuilding anything.

Contrast this with the Dockerfile norm, where each team picks a base image, a patch cadence, and a layer stack of its own. A foundation running fifty agents from fifty hand-rolled images is running fifty unique attack surfaces in fifty unknown patch states. These snowflake images are individually small risks and collectively a large one, because the attacker only needs the worst of them. The buildpack model removes the variation: one OS lineage, one patching authority, droplets that are reproducible and attributable. Broadcom's Tanzu 10.4 agent foundations release makes this same argument, describing trusted buildpacks as the mechanism that keeps agent containers automatically patched and verified.

Uniformity would matter less if remediation were slow, and this is the half of the buildpack argument that operations teams feel daily. Because the platform owns every layer, a fix at any of them rolls out without touching application code or taking the agent offline: a rolling restage replaces containers instance by instance behind the route, and users mid-conversation never see it. The cadence of vulnerability disclosure has outgrown quarterly patch windows; an architecture where "patch everything the agent runs on" is a routine, zero-downtime platform operation is the difference between reacting in hours and scheduling a change board.

This is the operating model Pivotal codified years ago as the three Rs of cloud-native security, and it applies to agents without modification. Repair: patch the stemcell, rootfs, or buildpack centrally and restage the fleet as soon as a fix exists. Repave: let BOSH continuously rebuild the underlying VMs from known-good stemcells, so anything an attacker managed to place on a machine has a lifespan measured in hours. Rotate: cycle credentials frequently, which the binding model makes mechanical since no credential is hard-coded anywhere a rotation could miss. The three Rs were designed to turn infrastructure from a static target into a moving one. An agent runtime inherits that motion for free by being an ordinary app on the foundation.

The reason to insist on all of this for agents specifically, rather than treating it as general hygiene, is where frontier models are taking offensive capability. Project Glasswing, a Linux Foundation initiative in which Broadcom participates alongside Anthropic, AWS, Apple, Cisco, Google, Microsoft and others, has shown publicly that frontier models can find high-severity vulnerabilities in widely used code. That capability is being pointed at defense, and the same class of capability raises the bar for what a production runtime must withstand. A fleet of inconsistent, stale container images is exactly the terrain such capabilities exploit fastest, and it is terrain the buildpack model simply does not present. Running an AI agent on the most disciplined supply chain available is not caution theater; it is matching your runtime posture to the threat model your own workload introduces.

5.2 Credentials

The agent holds no secrets of its own; section 4.2 and Figure 2 covered the mechanics, so here it is enough to state the consequences. The Anthropic API key arrives through a user-provided service or CredHub binding and surfaces only in the container's environment at runtime; tool credentials for MCP servers arrive the same way. Nothing is baked into the droplet, committed to git, or shared between applications, and Tanzu Platform's structural secrets isolation means one agent cannot read another's bindings even on the same foundation. Rotation is a rebind and restage, owned by the platform team.

Identity for tool calls deserves its own sentence, because it is where many agent deployments quietly fail audits. In this architecture the human's identity travels with the request: the user authenticates against Tanzu SSO, the MCP Gateway issues per-user tokens over OIDC, and the harness attaches the requesting user's token to every MCP connection their session makes. Tool actions are attributable to people, not to a shared service account with super-user reach.

5.3 Network

The foundation's posture is deny by default. The agent's container can reach api.anthropic.com because an operator wrote that egress rule, its bound services because bindings create the path, and nothing else. There is no general outbound internet from the agent loop; when the agent needs an external system, the path runs through an MCP server that the gateway fronts, where it is authenticated, authorized, and logged. This inverts the usual agent-framework default, where the process can call anything the host can route to. A prompt-injected agent on this foundation that decides to exfiltrate data discovers it has nowhere to send it.

5.4 The tool surface and the audit trail

The innermost layer is the SDK's own permission system, configured by the harness. The base tool set is read-only (Read, Glob, Grep, WebSearch, WebFetch, Skill); mutating tools like Bash, Edit, and Write are withheld unless explicitly enabled, and hooks can interpose organization-specific checks on any tool call before it executes. The agent's standing instructions live in CLAUDE.md and its procedures in skills, both version-controlled, both reviewable in a pull request like any other policy change.

Every layer above feeds one place: the record of what happened. The gateway writes a structured audit log of every tool invocation with the acting user attached; the platform aggregates application logs and the harness streams each tool call to the UI as it happens. When someone asks what the agent did, the answer is a query, not an investigation.

6. Data and Memory Architecture

The agent accumulates three kinds of state, and the architecture gives each a deliberate home rather than letting them collect in the container's ephemeral disk.

Conversational memory, the facts an agent learns and should retain across sessions, lives in Postgres behind an MCP memory server. The server exposes save and recall operations as ordinary tools, scopes facts per user or conversation, and resolves the agent's identity server-side from its binding, so tenant isolation is enforced where the data lives rather than trusted to the model's good behavior. The POC demonstrates this with its tanzumem server; the pattern generalizes to any memory service that speaks MCP.

Retrieval state, the embeddings that let the agent ground its answers in private documents, lives in the same Postgres through pgvector. Co-locating vectors with operational data is a deliberate trade. A dedicated vector database would add recall performance at very large scale, and it would also add a system, a credential set, a backup regime, and a failure mode. For the corpus sizes most enterprise agents actually retrieve over, pgvector inside an instance the platform already manages is the better default, and because retrieval reaches the agent through MCP, swapping the backend later does not disturb the harness.

Working state, the files an agent reads and writes mid-task plus the skills that define its procedures, lives on the NFS volume mounted into every instance. This is what makes the agent horizontally scalable in practice: any instance can pick up any session and see the same workspace, and a skill updated on the share is live for the whole fleet without a restage. Session transcripts themselves persist through the SDK's session mechanism, keyed by session ID, which is how a conversation survives both instance restarts and scale events.

Classify before you bind, and the data story stays clean: nothing the agent learns, retrieves, or writes leaves the foundation, because every store it touches is a bound platform service. Model inference sees conversation context in transit and retains nothing across calls. That sentence, verified against the architecture diagram, is most of a data-residency review.

7. Day-2 Operations

The operational argument for this architecture is that there is almost nothing new to say, and what follows is deliberately boring.

The agent scales like an app because it is one. `cf scale -i` adds instances behind the route; the NFS workspace and Postgres-backed sessions mean added instances are interchangeable. Deployments are blue/green with two routes, the same procedure the platform team already runs, and a bad agent release rolls back by remapping a route. Health is the `/health` endpoint under the platform's HTTP health checking, so a wedged agent process is recycled without a human noticing.

Observability comes from three streams that already exist. Application logs flow through the foundation's log aggregation into whatever the SRE team already watches; the harness emits each tool call as it happens, so the logs read as a narrative of agent behavior, not just request lines. The gateway's audit log covers the governance view. Platform metrics (container CPU, memory, request latency) need no agent-specific instrumentation.

Two genuinely new operational levers exist, and both are configuration. Model governance: the harness reads an allowlist of permitted model IDs and a default from the environment, so cost and capability decisions are an env var change, made by operators, not a code path owned by developers. Behavior updates: because skills and CLAUDE.md live on the shared volume, tuning what the agent knows and how it acts is a file change with a pull-request trail, deployable without a restage. The day a skill update misbehaves, reverting it is `git revert` plus a file copy.

Costs are the one area that needs new watching. Token consumption is the agent's marginal cost, it scales with conversation length and tool verbosity, and the platform's request metrics do not capture it. Pull usage from the Anthropic console or API into the existing dashboards early, set per-environment budgets, and let the model allowlist enforce the ceiling: routine traffic on a smaller model, the expensive one reserved for the workloads that justify it.

8. Getting Started

The distance from an empty space to a running agent is shorter than the architecture diagram suggests, because most of the boxes in Figure 3 are marketplace services that already exist on the foundation. What follows is the shape of a first deployment, drawn from the POC; the repository's README carries the full detail.

The platform team prepares the space once. The Anthropic API key goes into a user-provided service so it never touches a manifest or a repository; an SSO instance, a Postgres instance, and an NFS volume are created from the marketplace; MCP servers for whatever systems the agent should reach are deployed and registered with the MCP Gateway. None of these steps is agent-specific work. They are `cf create-service` and `cf cups`, the same provisioning grammar the team uses for every application.

The application team's path is a clone, a build, and a push. The manifest declares the buildpacks, the bindings, and the health check; `cf push` stages the droplet and the route goes live. A smoke test against `/health` and a first conversation through the chat UI confirm the loop end to end. From a standing start on an existing foundation, this is an afternoon, and the second agent is faster than the first because the space preparation amortizes.

Extension follows three grooves, in increasing order of ceremony. Teaching the agent a new procedure is a Markdown file dropped into the skills directory on the shared volume, live immediately, no deploy. Giving the agent a new system to act on is an MCP server registered with the gateway plus a binding, no harness change. Changing the harness itself (a new UI affordance, a different permission policy) is the only path that involves code, and it ships like any app change. Teams should expect the first groove to carry most of the traffic; an agent's value compounds through its skills faster than through its plumbing.

Three lessons from the POC are worth carrying into a first production deployment. Start the tool surface read-only and widen it deliberately; the base tool set answers most questions, and every mutating tool added should correspond to a workflow someone actually asked for. Put the behavioral contract in CLAUDE.md under code review from day one, because it is much easier to keep policy in version control than to move it there later. And wire token usage into a dashboard before the pilot grows, not after the first surprising invoice.

9. Conclusion

The question this paper set out to answer is where the enterprise agent loop should live. The answer it defends: on the platform you already operate, with model inference as the only thing you buy from outside the boundary.

The pieces required are not speculative. The Claude Agent SDK ships the production loop as a library. Tanzu Platform supplies the runtime containment (a buildpack supply chain controlled from stemcell to droplet), the governance machinery (SSO identity, the MCP Gateway's per-user tokens and audit log), and the managed state (Postgres with pgvector for memory and retrieval, NFS for skills and working files). The POC demonstrates the assembly end to end, and Broadcom's agent foundations direction in Tanzu 10.4 indicates the platform is investing further along exactly this line.

What makes the architecture durable is that it added no new operational concepts. Every hard problem an agent raises (whose credentials, which network paths, what audit trail, how to patch, how to roll back) was answered by a mechanism the platform team already runs. The agent is an app. That sentence, more than any individual feature, is the reference architecture.

9.1 Resources

Proof of concept repository: github.com/0pens0/tanzu-claude-agent (live deployment linked in the README)

Claude Agent SDK documentation: code.claude.com/docs/en/agent-sdk/overview

Tanzu Platform 10.4 and agent foundations: Broadcom announcement and techdocs (AI Services 10.4: MCP Gateway architecture, audit log schema, SSO/OIDC integration)

Project Glasswing: Linux Foundation announcement

The Three R's of Enterprise Security: Rotate, Repave, and Repair

---

"Stop building the platform.
Start building the agents that will transform the business."

Appendix A: Brokering Anthropic Access Through AI Services

The reference architecture in Section 4 treats the Anthropic API key as a user-provided service: an operator drops the key into a UPS once, the harness reads it from `VCAP_SERVICES`, and nothing else touches it. That pattern is deliberately minimal, and it is what the POC runs today. Tanzu Platform's AI Services tile offers a second path worth documenting separately, because it extends the same "credentials become service bindings" argument from Section 4.2 to the model connection itself, not just to tools and data.

A.1 The pattern

AI Services lets an operator register Anthropic as an off-platform model integration from the tile's marketplace configuration, the same mechanism used for any other externally hosted model. Once registered and added to a plan, Anthropic access is published to the Cloud Foundry Marketplace like Postgres or an NFS volume. A development team provisions a service instance against that plan and binds it to the harness with `cf bind-service`, and credentials surface through `VCAP_SERVICES` as a proxy endpoint and a short-lived key rather than a static secret the operator pasted in once.

The distinction from the main pattern is who owns the credential's lifecycle. A UPS holds whatever the operator put there until someone rotates it by hand. A marketplace-brokered instance inherits the platform's ordinary service lifecycle: rotation is an unbind and rebind, access is scoped by CF org and space the same way every other bound service already is, and the model connection shows up in the same plan-based governance view as the rest of the AI Services catalog. For an organization already running Postgres, NFS, and MCP servers through this broker pattern, brokering the model connection the same way is one fewer exception to explain to an auditor, not a new mechanism to introduce.

AI Services' AI Server middleware also sits in front of brokered connections, which is relevant beyond credential handling: load balancing and rate limiting happen centrally, at the platform layer, rather than being something each harness team reinvents per application.

A.2 When to use which

The direct UPS pattern in Section 4 is the one to reach for first: it is what the POC demonstrates, and it satisfies the paper's core claim that model inference is the architecture's one external dependency with nothing else in between. The AI Services-brokered pattern is worth evaluating for organizations that have already standardized model governance through AI Services for other providers and want Anthropic access to sit inside that same catalog rather than living as a one-off UPS.

vmware® by Broadcom

Item No: Running Claude Agents on Tanzu Platform 7/26