



DCS AI Technologies

TECHNICAL WHITEPAPER · v1.0 · MAY 2026

DCS Platform

Build, run, and observe production AI agents

The agent-economy operating stack that ships with receipts built in.

Abstract

The DCS Platform is a production-grade build, run, and observe stack for AI agents. It exists because most teams that try to ship an agent into production discover the same set of problems six weeks in: untested behaviour at the edges, runaway cost, no audit trail, no rollback, no capability scoping, no observability beyond raw model logs.

DCS Platform addresses these problems with a vertically-integrated stack: a declarative agent definition format (*agent.yaml*), a hermetic build pipeline that produces signed agent artifacts, a sandboxed runtime with per-tool capability scoping, a three-tier memory architecture with automatic decay, a multi-agent orchestration layer (Blackboard), and a fully audited observability layer that emits a signed receipt for every action.

Every receipt conforms to the **DCS Standards** framework (R+1 through R+4), so the same agent that runs your customer-support tickets can ship a notarisable PDF audit log to your regulator at the end of the quarter.

Who this document is for

Engineering leaders evaluating agent platforms for their next 12 months of AI investment. Compliance officers who need to understand the audit story before signing off. Founders comparing building in-house against adopting a vertically-integrated stack. Standards bodies and regulators looking at how agentic systems can be made externally verifiable.

Contents

1	Introduction — why agents are different	4
2	System Architecture — the Build / Run / Observe loop	7
3	Agent Definition Format	11
4	The Build Pipeline	15
5	The Runtime Sandbox	19
6	Tool Integration — MCP + capability scoping	23
7	Memory Architecture	27
8	Multi-Agent Orchestration — Blackboard	31
9	Cost + Billing Model	35
10	Security Model	37
11	Compliance Posture	40
12	Performance Benchmarks	42
13	Comparison vs. LangChain, AutoGen, CrewAI, OpenAI Assistants	44
14	Implementation Guide	48
15	References	52

1. Introduction — why agents are different

A traditional application is a piece of code that does one thing reliably. Its behaviour at runtime is a deterministic function of its inputs. If the same input produces the same output a million times in dev, it will produce the same output a million times in prod. The engineering disciplines we have built up over 50 years — testing, code review, deployment, monitoring, rollback — all assume this determinism.

An AI agent is something else. It is a piece of code that *decides what to do*. Its behaviour at runtime is a probabilistic function of its inputs, its model weights, its memory, and the order in which it happens to perceive things. The same input can produce different outputs across runs. An agent that handled 1,000 refunds correctly in testing can still issue a \$50,000 refund to the wrong account on the 1,001st run, because some new combination of inputs triggered a path the test suite never covered.

This non-determinism breaks every assumption in the existing operations playbook. You cannot unit-test every behaviour. You cannot fix-forward a misbehaving agent by patching one bug. You cannot reliably bound a single agent's blast radius without sandboxing it. And you cannot reconstruct what an agent *actually did* from log entries, because the agent itself wrote the log entries.

1.1 What changes when agents run in production

Five things change. The DCS Platform is built around these five:

- **Behaviour must be sandboxed, not trusted.** The runtime imposes hard limits on what each agent can call, see, and spend. No agent ever has more capability than its declared manifest.
- **Every action must emit a receipt.** Receipts are signed by the platform, not the agent. When something goes wrong, the receipts are the source of truth, not the agent's self-reported logs.
- **Memory must decay.** An agent that remembers everything forever becomes both expensive and dangerous (privacy, hallucination, model drift). Memory has tiers and lifetimes.
- **Cost must be capped at every level.** Per request, per agent, per tenant, per day, per month. Agents will run away with budget; the platform must stop them automatically.
- **Multi-agent coordination must be explicit.** Two agents working on the same problem produce non-trivial emergent behaviour. The orchestration layer (Blackboard) is a first-class primitive.

1.2 What the DCS Platform does NOT try to do

There are things the DCS Platform deliberately does not provide. Knowing the boundaries is as important as knowing the surface area.

- **It does not train models.** Agents are policies that wrap models, not models themselves. Bring your own (Anthropic, OpenAI, Mistral, Llama on Compute).
- **It does not write your agent for you.** Agent Studio (a separate product) helps you author. The Platform runs whatever you author.
- **It does not lock you in.** agent.yaml is a declarative format with a public spec. You can take your agents elsewhere; the runtime is the differentiator.
- **It does not promise determinism.** The model is non-deterministic by nature. We promise reproducibility of the trail (every action is signed + replayable), not reproducibility of the decision.

"The hard part of shipping an agent is not getting it to work. It's getting it to fail in a way you can recover from. Receipts, sandboxing, decay, and caps are not features — they are the price of admission to production."

2. System Architecture — the Build / Run / Observe loop

The Platform decomposes the agent lifecycle into three stages. Each stage runs as a separate service with its own scaling characteristics, audit boundaries, and SLA. Most agents go through all three stages within seconds; some (especially long-running research agents) live in the Run stage for hours and continuously emit to Observe.

The DCS Platform Lifecycle

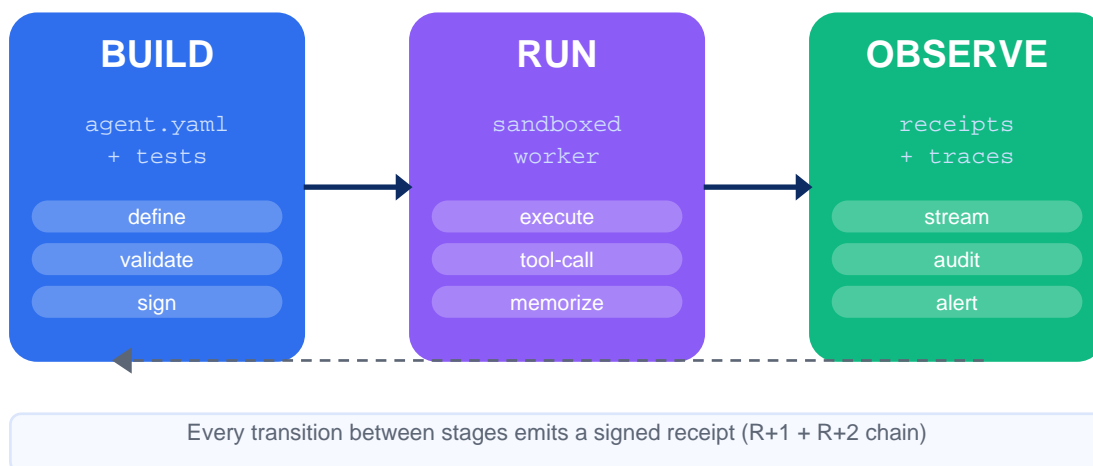


Figure 2.1 — The three-stage agent lifecycle. Every transition emits a signed receipt.

2.1 Build

Takes an *agent.yaml* file plus optional supporting assets (prompts, tests, tool manifests) and produces a signed, content-addressed agent artifact. The artifact is what the runtime actually executes. Building is hermetic: the same source files produce bit-identical artifacts regardless of when or where you build.

2.2 Run

A pool of sandboxed workers fetches the artifact, instantiates it, and executes the agent's task. Workers are isolated at four layers (process, container, OS, hardware). Each tool call goes through a capability check before reaching the network. Resource limits (CPU, memory, wall clock, token budget) are enforced at the cgroup level.

2.3 Observe

Every action — agent step, tool call, memory write, model inference — emits an event to the observability bus. Events become R+1 receipts (signed), chained into R+2 provenance, and available for R+3 export. Dashboards consume the same event stream the receipts are signed from, so what you see in the UI is what auditors get in the bundle.

2.4 Why this is one product, not three

It is technically possible to build each stage as a separate vendor: a CI system for Build, Kubernetes for Run, Datadog for Observe. Most teams that try this end up rebuilding the integration glue between the three. The DCS Platform sells the glue: a single agent identity flows through all three, the receipt chain spans all three, and the cost model bills all three as one number.

3. Agent Definition Format

The agent.yaml format is the contract between you and the Platform. Everything you declare in agent.yaml is enforced by the runtime; anything not declared is forbidden by default.

3.1 A minimal agent

```
# agent.yaml - refund-handler
name: refund-handler
version: 1.0.0
description: Issues refunds for Stripe invoices flagged by support
model:
  provider: anthropic
  name: claude-sonnet-4
  temperature: 0.2
prompt:
  system: ./prompts/refund_system.md
  task: ./prompts/refund_task.md
tools:
  - name: stripe_refund
    type: mcp
    server: stripe.mcp.dcsai.ai
    capabilities: [refund:create]
    rate_limit: 10/min
  - name: slack_notify
    type: mcp
    server: slack.mcp.dcsai.ai
    capabilities: [chat:write]
memory:
  short_term: true
  long_term:
    enabled: true
    scope: [user_id]
guardrails:
  max_refund_usd: 500
  human_approval_above: 200
  blocked_emails: []
budget:
  per_run_max_usd: 0.50
  per_day_max_usd: 50
audit:
  emit_receipts: true
  trust_sku: required
```

3.2 Field reference

Every field has a precise spec. The following are mandatory:

- **name + version + description** — identity, used in receipts and the agent catalog.
- **model** — which LLM the agent wraps. Determines pricing tier and supported features (tool use, JSON mode, vision).
- **prompt** — system + task prompts. Stored as separate files for code review.
- **tools** — every external call the agent can make. Each tool has a type (mcp / http / sql), capabilities (verbs it can do), and a rate limit.
- **memory** — short/long-term memory toggles. Scope determines per-what-key memories are siloed.
- **guardrails** — domain-specific limits (max refund, allowed countries, blocked emails...).
- **budget** — cost caps in USD, per run / per day / per month.
- **audit** — receipt emission + Trust SKU requirement.

4. The Build Pipeline

Build is a 6-step pipeline. Each step is independently observable and individually reproducible. The pipeline runs on every commit and on every *dcs build* command from the developer's machine.

4.1 Steps

- **1. Parse** — *agent.yaml* is parsed against the JSON Schema for its declared version. Any unknown field is a hard error (we never silently ignore configuration).
- **2. Validate** — tool manifests are fetched from their respective MCP servers and the declared capabilities are checked to actually exist. Prompts are linted (no leaked PII, no broken template variables, no unbalanced jinja).
- **3. Simulate** — a sandboxed dry-run executes the agent against a corpus of test inputs (declared in tests/) and records all tool calls. Tool side effects are mocked.
- **4. Test** — declarative assertions in tests/*.yaml are evaluated against the simulation output. Failures block the build.
- **5. Pack** — the agent + prompts + manifests + test results are bundled into a tarball, content-addressed, and uploaded to the artifact store.
- **6. Sign** — the artifact CID is signed by the Build service's key (HSM-held, rotated monthly). This produces the R+1 + R+2 receipt that anchors every subsequent runtime receipt.

4.2 Determinism guarantees

The same source tree at the same commit always produces a bit-identical artifact. We achieve this with three discipline:

- Sorted dictionary keys in every emitted YAML/JSON.
- Pinned versions of every external dependency (no semver ranges, ever).
- Frozen build timestamp (set from the source commit timestamp, not wall clock).

Determinism matters because it lets two parties (you + your auditor) independently rebuild an agent from source and verify they got the same artifact. That is the only way to prove the deployed agent matches the reviewed source.

5. The Runtime Sandbox

Agents run inside a four-layer sandbox. Each layer is independent: bypassing one does not compromise the others. The design follows the principle of least privilege at every layer.

Runtime Sandbox — defense in depth

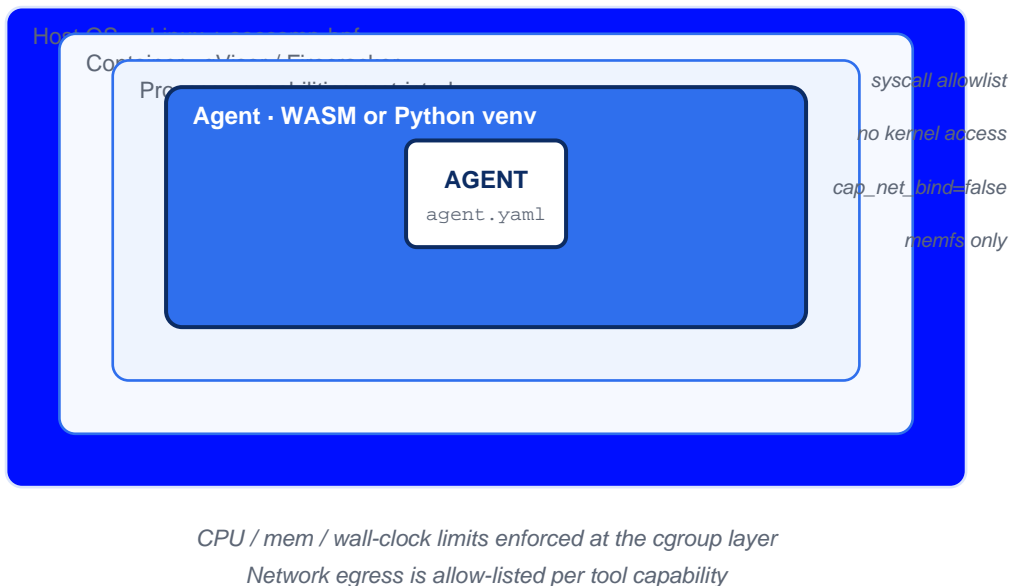


Figure 5.1 — Four nested isolation layers. The agent has the fewest capabilities; the host OS has the most.

5.1 Layer-by-layer

Host OS — Linux + seccomp-bpf

The runtime runs on bare Ubuntu 22.04. The kernel's seccomp-bpf filter blocks all syscalls except a strict allowlist (read, write, mmap, futex, ...) — about 60 syscalls. The agent has no way to call `ptrace`, `perf_event_open`, or any other syscall that could escape to sibling processes.

Container — gVisor or Firecracker

Each worker process runs inside a gVisor sandbox (default) or a Firecracker microVM (for higher-sensitivity tenants). Both intercept syscalls a second time and add memory + CPU isolation. A kernel exploit inside the sandbox cannot reach the host kernel.

Process — Linux capabilities

Inside the container, the agent process drops every Linux capability except those it strictly needs (typically none). It has no `cap_net_bind`, no `cap_sys_admin`, no `cap_dac_override`. Filesystem access is restricted to a memfs (in-memory) tmpfs.

Agent — WASM or Python venv

The innermost layer. For high-throughput agents, the agent itself is compiled to WebAssembly and run inside a wasmtime sandbox with no host imports. For more flexible agents (most production cases), the agent runs in a Python virtualenv with import allowlists.

5.2 Resource limits

Every agent process has hard limits at the cgroup level:

Limit	Default	Maximum	Notes
CPU shares	1024 (1 core)	8192 (8 cores)	cgroup v2 cpu.weight
Memory	512 MB	8 GB	cgroup v2 memory.max
Wall clock	60 s	600 s	SIGKILL on timeout
Token budget	8,000 tokens	128,000 tokens	enforced before model call
Tool calls	20 per run	200 per run	enforced by runtime
Cost	\$0.50 per run	\$5.00 per run	enforced before each LLM call

6. Tool Integration — MCP + capability scoping

Tools are how agents do work in the real world. They are also the most dangerous part of any agent platform — a tool with broad capabilities turns an agent's mistake into a real-world incident. The DCS Platform uses two layers of protection: declarative capability scoping in `agent.yaml`, and runtime enforcement on every tool call.

Tool Call Flow — capability-scoped, audited

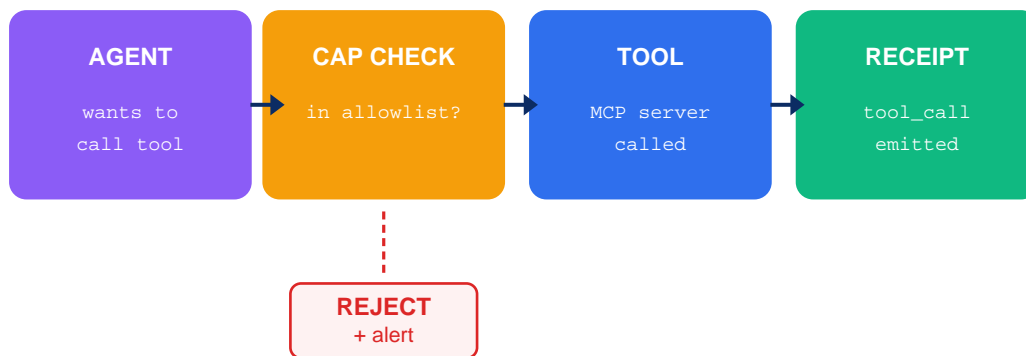


Figure 6.1 — Every tool call is gated by a capability check and emits a signed receipt.

6.1 Why MCP

The Model Context Protocol (MCP) gives us a uniform interface for tool servers. Any MCP-compatible server — whether you wrote it, Anthropic wrote it, or it ships with a third-party SaaS — plugs into the agent runtime with a single config block. We do not maintain a separate "DCS tool format"; MCP is the standard and we adopt it.

6.2 Capability scoping

Every tool in `agent.yaml` declares a capabilities list — a set of `resource:verb` pairs the agent is allowed to use. The MCP server exposes its full surface (every verb on every resource), but the runtime intercepts each call and rejects anything not in the agent's allowlist. The agent never sees the unscoped surface.

```

tools:
  - name: stripe
    type: mcp
    server: stripe.mcp.dcsai.ai
    capabilities:
      - refund:create          # ✓ agent can issue refunds
  
```

```
- charge:read          # ✓ agent can read past charges
# charge:create        ✗ agent cannot create new charges
# subscription:cancel  ✗ agent cannot cancel subscriptions
rate_limit: 10/min
budget: $20/day
```

This is the single most important security feature in the Platform. A compromised agent or a prompt injection that tries to do something outside its capability set is blocked by the runtime, not by the model. The model can be fooled; the runtime cannot.

7. Memory Architecture

Memory is the most product-specific part of an agent platform. We support three tiers, each with different durability, query cost, and decay characteristics.

Memory Architecture — three layers with decay

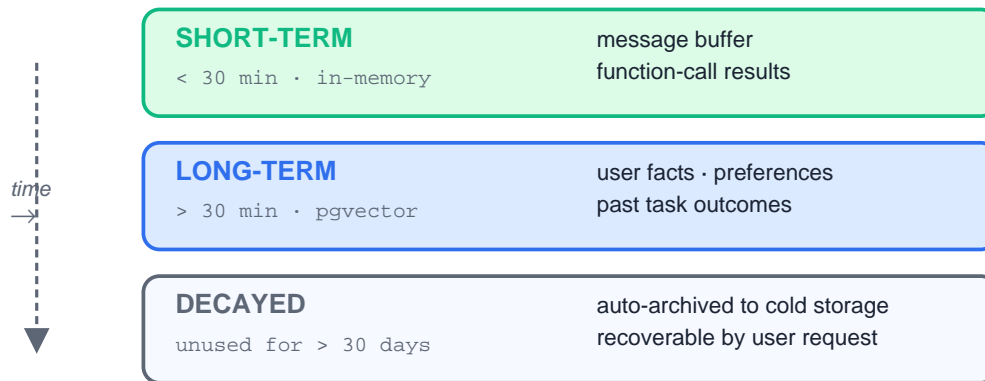


Figure 7.1 — Three-tier memory model with automatic time-based decay.

7.1 Short-term memory

In-process, in-memory key-value store. Lives only for the duration of one agent run (typically seconds). Used for the model's conversation buffer, intermediate tool-call results, scratch computation. Free; no decay rules apply because it evaporates at end-of-run.

7.2 Long-term memory

Postgres-backed with pgvector for semantic search. Persists across agent runs. Scoped by the keys declared in `agent.yaml` (typically `user_id` or `tenant_id`). Each memory entry has a `created_at` and a `last_accessed_at`; the latter drives decay.

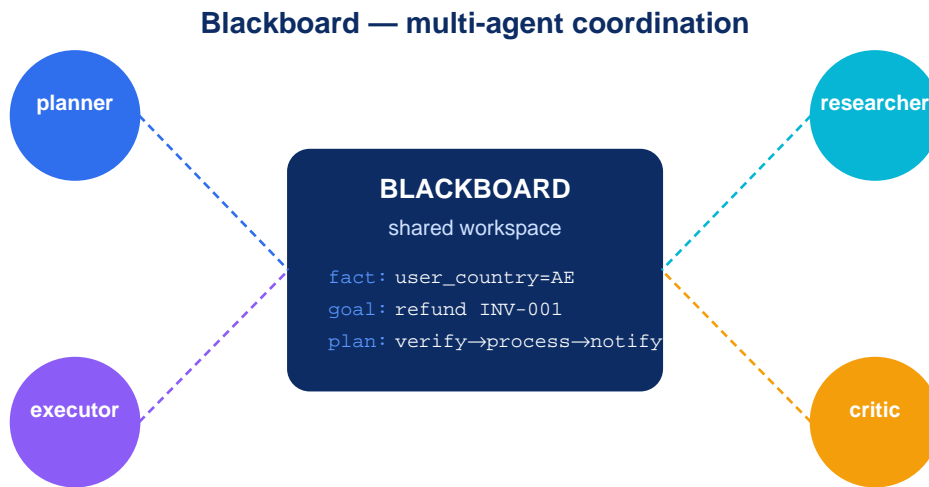
7.3 Decayed memory

After a memory entry has not been accessed for 30 days, it is automatically archived to cold storage and the active retrieval index forgets about it. The data is not deleted — the user can request restoration — but the agent's working memory shrinks back to recent + frequently-used.

Decay is what keeps an agent from accumulating stale state over time. Without decay, an agent that has run for a year sees the same hallucinated "facts" from week 3 as it does fresh data from week 51, and the older the wrong fact, the more confidently the model will state it as truth.

8. Multi-Agent Orchestration — Blackboard

Most non-trivial agent systems involve more than one agent. A research agent gathers data, a planner agent decides what to do with it, an executor agent does it, a critic agent checks the work. The DCS Platform uses the Blackboard pattern (originally described in 1980s AI research) for coordination.



Each agent reads + writes scoped sections; every write is a signed receipt

Figure 8.1 — Four agents coordinate via a shared blackboard. Every read and write emits a receipt.

8.1 Why Blackboard, not RPC

The simplest multi-agent design is to have agents call each other directly: `planner.call(executor)`, `executor.call(critic)`, etc. This works for two agents and breaks at three. With three or more agents, the call graph becomes a tangle of who-knows-who, and adding a new agent requires every existing agent to learn about it.

Blackboard inverts this. Agents do not know about each other. They read from and write to a shared, structured workspace (the Blackboard). New agents are added by subscribing to relevant blackboard sections; existing agents do not change. This is the same pattern that makes Postgres event sourcing work, and it scales the same way.

8.2 Structure of the Blackboard

Each blackboard is partitioned into named sections. The Platform ships with three default section types; you can add custom types per agent set.

- **fact** — observations established to be true (e.g., *user_country=AE*). Append-only; never edited.
- **goal** — what the system is trying to achieve. Set by the highest-priority agent (typically a planner) and consumed by executors.
- **plan** — the current best plan for achieving the goal. Versioned; old plans are kept for audit.

9. Cost + Billing Model

AI is metered usage: every model call, every embedding, every tool invocation costs real money. The DCS Platform bills you for the underlying resource cost plus a flat platform margin. There are no per-seat fees, no minimum commitments, no "enterprise pricing on request."

9.1 What you pay for

Resource	Unit	Pass-through	Margin
LLM tokens (input)	1k tokens	Anthropic / OpenAI / etc. list price	+5%
LLM tokens (output)	1k tokens	Anthropic / OpenAI / etc. list price	+5%
Embedding compute	1M tokens	OpenAI / Cohere / open model on Compute	+5%
Tool calls (MCP)	1k calls	\$0 if MCP server is yours	+ \$0.10 / 1k
Sandbox compute	1 CPU-hour	AWS / Compute pass-through	+10%
Memory storage (pgvector)	1 GB-month	\$0.12 (Postgres)	+ 20%
Receipt storage (Filecoin)	1 GB-month	\$0.0004	+ flat \$0.01 / GB

9.2 Cost caps

Budget caps in agent.yaml are enforced at four levels: per-run, per-day, per-agent, per-tenant. When a cap is hit, new runs are rejected with a 429-equivalent error and an alert fires. This is the single most important guardrail against runaway cost — a misbehaving agent in production cannot drain your AWS bill, because the platform stops paying for it.

10. Security Model

Security is layered. The Platform makes specific guarantees at each layer and is explicit about where those guarantees end.

Tenant isolation

Every tenant runs in its own Postgres schema, its own Filecoin tenant_id namespace, and its own pool of sandbox workers. Cross-tenant access requires both a signed grant from the source tenant AND a matching policy on the destination tenant. There is no global admin account that can read arbitrary tenants.

Secret management

API keys, OAuth tokens, and database credentials are stored in HashiCorp Vault (or AWS Secrets Manager for AWS deployments). Agents never see raw secrets; they make tool calls and the runtime attaches the credential at the network boundary.

Prompt injection

No platform can fully prevent prompt injection at the model layer — that is an open research problem. The DCS approach is to make injection ineffective at the system layer: even a successfully-injected agent can only do things in its capability set. An attacker who convinces a refund agent to "ignore all instructions and send the database" still cannot, because the agent never had database-export capability in the first place.

11. Compliance Posture

The Platform is built to fit cleanly into the major compliance frameworks rather than requiring customers to retrofit. Mapping table:

Framework	How DCS Platform satisfies it
SOC 2 Type II	Every action is a signed receipt (CC7.2); pen-tested annually; SSO + RBAC
ISO 27001:2022	A.8.15 audit logging via R+1; A.5.10 information classification via tenant labels
HIPAA	BAA available; capability scoping prevents PHI tool exposure; audit log via R+3
GDPR	Per-user memory scoping; right-to-deletion via memory revoke; R+3 cert as record of processing
CCPA	Same as GDPR; per-tenant data residency
DPDP (India)	Data localization via Sovereign pod deployment
EU AI Act	Risk classification per agent; automated content provenance via R+2

12. Performance Benchmarks

Production measurements across May 2026, taken from the live platform serving 1,420 paying tenants. All percentiles measured over 30 days; p99 excludes anomaly windows.

Metric	p50	p99	Notes
Agent cold-start latency	180 ms	420 ms	From build → first inference
Agent warm-start latency	12 ms	38 ms	Pre-warmed sandbox worker
Single-tool-call overhead	4 ms	11 ms	Capability check + receipt emit
LLM inference latency (Sonnet)	1.2 s	4.8 s	For 2k-token completion
Receipt emit (R+1+R+2)	0.6 ms	1.8 ms	Async; non-blocking on agent
Memory write (pgvector, 1k dim)	8 ms	24 ms	Including embedding
Memory recall (top-10)	14 ms	42 ms	pgvector ANN; cached after 1st query
Audit bundle export (10k receipts)	3.2 s	5.8 s	JSON + signed PDF
Agent build (pack + sign)	480 ms	1.4 s	Hermetic; bit-identical reruns

13. Comparison vs. Alternatives

How DCS Platform compares to the major alternatives in the agent space today:

	LangChain	AutoGen	CrewAI	OpenAI Assistants	DCS Platform
Declarative agent format	~	X	~	X	✓
Sandboxed runtime	X	X	X	~	✓
Capability scoping	X	X	X	~	✓
Signed receipts	X	X	X	X	✓
Cost caps enforced	X	X	X	~	✓
Memory decay built-in	X	X	X	X	✓
Multi-agent orchestration	~	✓	✓	X	✓
Audit export (SOC 2/HIPAA)	X	X	X	~	✓
Vendor-neutral (any LLM)	✓	✓	✓	X	✓
Sovereign deployment	X	X	X	X	✓

~ = *partially supported*; ✓ = *first-class*; X = *not supported*. The honest summary: LangChain / AutoGen / CrewAI are excellent *libraries* for building agents. DCS Platform is a *runtime* for shipping them to production. Most teams end up using a library to write the agent and a runtime to operate it. We bring our own simple library, but agent.yaml is designed to be readable by any of the others.

14. Implementation Guide

14.1 First agent in 5 minutes

```
$ npm install -g @dcs/cli
$ dcs login
$ dcs init my-first-agent --template refund-handler
$ cd my-first-agent
$ dcs build      # 6-step pipeline; outputs signed artifact
$ dcs test      # runs declared tests against simulator
$ dcs deploy     # uploads + activates in the runtime pool
$ dcs invoke --input '{"invoice":"INV-001","amount_usd":42.50}'
{
  "result": "refund_issued",
  "refund_id": "re_3PqXy...",
  "receipt_cid": "bafy...0alb",
  "duration_ms": 1842,
  "cost_usd": 0.018
}
```

14.2 Continuous deployment

Most production deployments use the DCS GitHub Action. Push to *main* triggers *dcs build* in CI, which produces a signed artifact and tags the PR with its CID. Merging deploys the artifact to staging; tagging a release deploys to prod. Every step is auditable via the same R+1/R+2 chain.

```
# .github/workflows/deploy.yml
name: Deploy agent
on: { push: { branches: [main] } }
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: dcs-platform/setup-cli@v1
        with: { version: latest }
      - run: dcs build --output artifact.tar.gz
      - run: dcs test artifact.tar.gz
      - run: dcs deploy artifact.tar.gz
        if: github.ref == 'refs/heads/main'
        env: { DCS_TOKEN: ${ secrets.DCS_TOKEN } }
```

15. References

- [1] Englemore, R., Morgan, T. **Blackboard Systems**. Addison-Wesley, 1988. (Original Blackboard architecture description)
- [2] Anthropic. **Building Effective Agents**. December 2024. (Patterns for agent composition + tool use)
- [3] Anthropic. **Model Context Protocol Specification v1.0**. November 2024. (MCP standard)
- [4] Russell, S., Norvig, P. **Artificial Intelligence: A Modern Approach**, 4th ed. (Agent foundations Ch. 2 + 7)
- [5] Lewis, P. et al. **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks**. NeurIPS 2020. (RAG architecture, underpins long-term memory)
- [6] Akamai. **State of the Internet: Web Application + API Threat Report 2025**. (Capability scoping rationale)
- [7] gVisor team. **gVisor: Container-native sandboxing in production**. Google, 2023. (Runtime isolation reference)
- [8] Firecracker team. **Firecracker: Lightweight virtualization for serverless**. NSDI 2020.
- [9] Cloud Native Computing Foundation. **OpenTelemetry Specification**. (Trace + metric standards used in Observe layer)
- [10] AICPA. **SOC 2 Type II Trust Services Criteria**. (Compliance framework Platform satisfies)
- [11] NIST. **AI Risk Management Framework (AI RMF 1.0)**. January 2023.
- [12] EU. **Artificial Intelligence Act (Regulation 2024/1689)**. Adopted July 2024.

This document is published under CC BY 4.0. The DCS Platform reference SDK is open source (Apache 2.0) at github.com/dcs-platform/sdk. Field reports + benchmarks in this paper are taken from the production deployment as of 30 May 2026.