SiloOS

The Agent Operating System for AI You Can't Trust

Scott Farrell

https://leverageai.com.au

scott@leverageai.com.au

SILO OS ARCHITECTURE - TECH PAGE

This is the techo page - skip for business folks

A security-first, privacy-first execution environment for AI agents. Intelligence is free to think, but every action is mediated through capability-checked proxies and short-lived authority tokens.

Security First	Privacy First	AI First-Class
The Padded Cell	Tokenisation	Atomic Agents

The Padded Cell: Each agent runs in isolation—its own Linux user, temp folder in RAM. The only way out is through the membrane: proxies that enforce keys, policies and tokenisation. Nothing unlogged, nothing unapproved, nothing persistent.

Privacy First: Agents operate in a tokenised reality. They never see raw PII. Want to send an email? Submit the template and customer token—the proxy hydrates and sends it. The agent never touches real data.

AI as First-Class: Agents are atomic folders using the Markdown Operating System pattern. Markdown files are operating instructions, not docs. Python provides tools for computation. The LLM handles reasoning. Everything is plain text—when something breaks, you can read the instructions, check the state, review outputs. No black boxes.

```
refund-agent/
  main.py
  tools.py
  playbook.md
  policy.yaml
Atomic. Inspectable.
Shippable.
```

Proxy Services: Agents talk only to proxies—database, email, SMS. Each request requires both base keys (agent's role capabilities) and task keys (this customer, this case). No direct Redis access; just Redis-like access where you throw your JWT at every interaction.

Router
The Kernel
Mints keys. Routes tasks.
Logs everything.

Event-Driven Router: The router isn't HTTP—it's an event-driven kernel on queues. Frontends fire events (TaskCreated, Escalate, Complete). The router consumes them, applies routing rules from a fixed transition map, mints task keys, emits new events. No agent-to-agent comms. No RPC tangle. Pure message flow.

Humans Are Agents Too: Same tools, same markdown instructions, same proxy constraints, same task keys. Route to human when: agent offline, request exceeds authority, customer gets nasty, or shadow-mode 1-in-N for continuous evaluation.

The Four Pillars

Base Keys
Stable capabilities per role. "Refund up to $500". Never contains PII.

Task Keys
Short-lived tokens for this customer, this case. Evaporates when done.

Tokenisation
Sensitive data stays in proxy. Agents work only with tokens.

Stateless Execution
main.py runs once, produces outputs, dies. No memory, no drift.

Tokenised data. Capability-bound actions. Stateless workers. Event-driven router holding the keys.
Horizontal by default—run more copies, no coordination, no locks. Stop trying to trust AI. Build the cell.

It's the Matrix in reverse. *Instead of humans plugged into a simulation, SiloOS plugs in the agents—and lies to them, politely and securely, about what "reality" is.*

AGENTS don't see the real database. AGENTS don't see real customers. AGENTS live inside a padded cell fed a tokenised world—handles, abstractions, shadows projected onto the cell wall. The membrane interprets what the agent *thinks* it's doing, maps it to safe actions, and enforces strict physics inside the simulation.

When an agent hesitates, we plug in a human. Same workspace, same tools—but the human knows the membrane is there. The agent just blinks behind its window, seeing whatever SiloOS allows. **In the Matrix, humans live in a simulation controlled by machines. In SiloOS, machines live in a simulation controlled by humans.**

THE TRUST FALLACY

Why every current approach to AI security shares the same fundamental flaw

Every approach to AI security currently deployed assumes we can make AI trustworthy enough to grant it access. This assumption isn't just optimistic—it's architecturally backwards. Trust is the wrong security model for an entity that writes its own code at runtime.

Walk into any enterprise IT department and you'll encounter four strategies: **alignment training** (a research problem, not engineering solution), **guardrail prompts** (security by obscurity), **human oversight** (defeats the point of AI speed), and **policy frameworks** (detect violations after the fact). Together, they create an illusion of security without preventing anything.

95%

of AI initiatives stall before reaching production

MIT's State of AI in Business 2025 reveals why: not lack of capability, but friction—security reviews, compliance checks, organizational change.

Why Current Approaches Fail

Alignment: Can be jailbroken
Prompts: Can be overridden
Oversight: Doesn't scale
Policies: Detect, don't prevent

Policies don't prevent—they document. After-the-fact auditing doesn't stop the data leak, the unauthorized refund, or the privacy violation.

Traditional Security vs. AI Agent Reality

Traditional Assumptions	AI Agent Reality
• You control the code	• AI writes code at runtime
• Behavior is deterministic	• Non-deterministic behavior
• Audit before deployment	• Can't audit what hasn't been generated

This is a fundamental incompatibility between security model and system. Traditional security evolved for deterministic systems. AI writes code at runtime that can't be pre-audited.

We trust humans through accountability and consequences. AI agents have none of these properties. The industry has anchored on the wrong question: "How do we make AI trustworthy?" This leads to endless security reviews and compliance theater.

The Trust Model Doesn't Fit

AI has no:

- Accountability (who do you fire?)
- Reputation to protect
- Meaningful consequences
- Organizational loyalty

The Right Question:

"How do we build systems where AI's trustworthiness doesn't matter?"

This reframes the problem from alignment (unsolved) to architecture (solvable). The system compensates through containment.

This is a fundamental shift from controlling the AI to controlling what it can access. One path leads to research labs and decade-long timelines. The other leads to production systems you can deploy next quarter.

THE PADDED CELL

Security architecture that makes AI trustworthiness irrelevant

The Prisoner Analogy

Genius-level intelligence. Profound insights. Completely untrustworthy. You need what they can do—but you can't let them out.

Solution: A padded cell. Not to torture, but to contain. Full capability, walled scope.

SiloOS is built on a radical premise: don't make AI trustworthy—make its trustworthiness **irrelevant**. The AI doesn't need to be safe if its environment is secure.

Traditional security restricts capability to be safe. SiloOS inverts this: **expand capability, restrict scope**. Give the agent everything it needs to work brilliantly—full LLM reasoning, rich tools, decision authority. But wall off the scope. It can work on what you give it. Nothing else.

The Four Pillars

The padded cell isn't a metaphor—it's an architecture. Four interconnected pillars turn theory into deployable infrastructure:

1. Base Keys

What the agent **type** can do. Role-based capabilities: `refund:max_$500` , `email:send`

2. Task Keys

What this specific **instance** can access. Scoped to customer, case, session: `customer:TOKEN_847a2` , `expires:20min`

3. Tokenization

Agent never sees real PII. Works with tokens: `[NAME_1]` , `[EMAIL_1]` —never "Jane Smith"

4. Stateless Execution

Each invocation starts fresh. No persistent memory. Context terminates when task completes. Clean slate.

These pillars are *composable*. Base keys define capability—once. Task keys scope data access—per invocation. Tokenization protects privacy—by default. Stateless execution prevents accumulation—always.

The result: security that scales O(1) with agent count, not O(n). The AI "literally **cannot**" exceed scope—not *shouldn't*, not *is told not to*. Architecture makes it impossible.

Zero Trust for AI

Traditional zero trust assumes you control the code. AI agents write code at runtime—emergent behaviors can't be pre-audited.

SiloOS adapts zero trust: Unique identity per agent. No implicit trust. Continuous verification based on *who*, *what*, and *when*.

Defence in Depth

SiloOS layers multiple independent controls. Any single failure doesn't compromise the system:

1. Container isolation + Linux capabilities
2. Network segmentation (agent → proxy only)
3. Key validation at proxy (JWT tokens)
4. Tokenization layer (PII never reaches agent)
5. Immutable audit logs
6. Stateless execution (context terminates)

Assume breach posture: *Design as if the AI will misbehave—because we can't prove it won't.*

Why This Matters

You know the security *principles*—least privilege, zero trust. You don't have the *pattern* for AI. Traditional security assumes you

BASE KEYS AND TASK KEYS

The architectural separation that makes SiloOS security work

The core innovation of SiloOS lies not in what it restricts, but in how it separates. Every security model before this conflated two fundamentally different questions: *What can this agent do?* And *what data can it touch?* SiloOS tears them apart.

Think about a customer service representative. They have a job description—what they're authorized to do. Process refunds up to $500. Send emails. Escalate when needed. But that job description doesn't grant them access to every customer record. When they log in, they get access to *this customer*, *this case*, *this conversation*.

The Independence Principle

Base Keys: What the agent *can do* (capabilities, limits, escalation rights)

Task Keys: What data it *can access* (customer token, case ID, session)

Both must be satisfied. Neither can compensate for the absence of the other.

The job description is the base capability. The case assignment is the scoped access. Base keys define agent authority. Task keys define data scope. Both must be satisfied for any action.

Why Separation Matters

Without Separation

- Access control becomes all-or-nothing
- Capability limits require data access to enforce
- Scaling requires duplicating access grants

With Separation

- Agent has capability but can't act without task key
- Task key grants access but can't enable actions without base key
- Job roles change independently of task assignments

The Security Matrix

Situation	Base	Task	Result
Normal operation	✓	✓	Allowed
Exceeds authority	✗	✓	Escalate
Wrong data scope	✓	✗	Rejected
Both missing	✗	✗	Rejected

Base keys are the job description. They encode what an agent *type* is authorized to perform. A customer service agent might have base keys for refunds up to $500, sending emails, escalating to a manager. A collections agent might have authority for higher refunds, payment plans, account suspension.

Task keys are the case assignment. They encode the specific customer, case, or session this particular invocation relates to. When a customer chat arrives, the router mints task keys for that interaction. Not for all customers. Just this customer, this case, this conversation. The keys are scoped, temporary, and expire when the task completes.

This separation is why SiloOS can grant agents powerful capabilities without risking broad data exposure. The refund agent might have authority to issue $500 refunds—but without a customer task key, it can't refund anyone. The capability exists in the abstract. The task key makes it concrete.

> *"The task keys are really just the customer token and the case ID. The functionality is in the agent. The $500 limit is on the agent base stuff."*

SiloOS uses JWT-style tokens for both key types—a well-understood, cryptographically sound, stateless pattern proven at scale. Recent academic research on "Agentic JWT" validates this approach, proposing extensions that cryptographically bind each agent action to verified user intent—exactly the pattern SiloOS implements. Both must be satisfied for any action. The agent must have the capability *and* the scoped access. Neither can substitute for the other.

CHAPTER 5: TOKENISATION

The Agent Never Sees Real Data

Privacy by Architecture

In SiloOS, agents work with tokenised representations like `[NAME_1]` and `[EMAIL_1]` —never touching actual PII.

The proxy holds the mapping and hydrates tokens only when actions execute. This satisfies GDPR, CCPA, and legal teams through architectural impossibility, not policy documents.

Privacy isn't a feature you bolt on—it's an architectural foundation. Large language models are phenomenal tools and phenomenal privacy risks. They may retain echoes of training data. Prompts can be logged, leaked. Third-party LLM APIs sit outside your security perimeter.

Between 2024 and 2025, employee data flowing into GenAI services grew 30× almost overnight. Traditional perimeter security fails because the perimeter has shifted—to browser windows and prompt interfaces where sensitive content is routinely shared.

Tokenisation replaces real PII with reversible tokens before any agent access. The agent reasons about customers, makes decisions—it just never sees actual names, addresses, or contact details.

Notice what's *not* tokenised: account balance, order counts. These are operationally necessary and not personally identifying. The agent needs to reason about whether a $247.50 balance justifies a refund. That context stays visible.

When the agent needs to take action—send an email, validate a phone number—the proxy hydrates the template with real data. The agent reasons about the customer without seeing who they are.

What the agent sees:

```
{
  "name": "[NAME_1]",
  "email": "[EMAIL_1]",
  "phone": "[PHONE_1]",
  "balance": 247.50,
  "orders": 8
}
```

Non-sensitive data (balances, counts) remains clear. PII is always tokenised.

Wells Fargo: 245M Interactions, Zero PII Exposure

Speech transcription locally — audio never leaves secure environment

Query routing internal — uses internal models

LLM receives anonymised context only — external model treated as untrusted

The Flow

Router sends task with customer token `tok_8f3k2`. Agent requests data. Proxy validates, retrieves, tokenises PII, sends tokenised structure. Agent processes without seeing identifiers. When executing—"Send refund email to `[EMAIL_1]`"—proxy hydrates and executes. Audit log shows tokens only.

Microsoft Presidio

Open-source PII detection. AI never sees real data. Compliance through architecture.

This is compliance revolution. When legal asks "how do we ensure AI doesn't misuse customer data?" the answer is architecture—the AI can't misuse data it never sees.

STATELESS EXECUTION

Each agent invocation starts fresh. No persistent memory between runs. No accumulated context from previous customers. No data leakage across sessions. This is both security architecture and operational elegance.

When you deploy AI agents in production, one of the most powerful decisions is whether to give them memory. Intuition says agents should remember previous conversations, past customers, and learn from experience.
SiloOS says the opposite. Agents start fresh, every time.
Stateless execution means no memory of previous tasks. Each invocation is completely independent. The agent processes it, returns a result, then the entire execution context—conversation history, intermediate attempts, error logs, temporary files—vanishes.

The Lifecycle

1. Task arrives
2. Keys issued (scoped)
3. Agent processes
4. Result returned
5. Context terminates
6. Fresh instance next

Security and Architecture Wins

Attack Containment

A compromised agent cannot:

- ✗ Access prior customer data
- ✗ Persist malicious code
- ✗ Accumulate credentials
- ✗ Build database knowledge

When an agent has no memory, entire classes of attacks become impossible. Customer A's data can't leak into Customer B's session because there's no session. If an agent extracts sensitive information, it vanishes when the task ends.
Horizontal scaling is trivial. Need more load? Spin up more instances. They don't coordinate, don't share state, don't step on each other's toes.
Debugging is reproducible. Grab the task inputs, keys, and agent code. Re-run. Same result every time. No mysterious state from five customers ago.

The Temp Folder Pattern

Agents can only write to a temporary folder that gets wiped when the task completes. Often RAM-based for speed and security.

RAM Disk
Fast, secure, auto-cleared on exit

Session Dir
Flexible for large files

Container FS
Maximum isolation

Sub-Agents and Context Evaporation

The **sub-agent** pattern: spawn a temporary agent for messy subtasks. It works in isolation, handles errors, accumulates state—then returns a clean artifact and terminates. Everything evaporates. Main agent receives only the result.

The Pattern

Example: Generate 8 images

Without sub-agents: 15,000 tokens of logs

With sub-agents: 600 tokens. Sub-agent's journey evaporates.

What Persists, What Evaporates

Business data (persistent): Customer records, case status, refund transactions. Written to the database through the proxy with valid task keys.
Audit logs (persistent): Every action—data accessed, keys used, actions performed. Logged centrally for compliance.
Agent state (ephemeral): Working memory, conversation history, intermediate files. Vanishes when the task completes.

THE MARKDOWN OPERATING SYSTEM

When an agent is just a folder

The Folder Philosophy

An agent is a *folder*. That folder contains everything it needs: entry point, tools, instructions, templates. Nothing more.

When you picture an AI agent, you might imagine thousands of lines of code, intricate state machines, elaborate frameworks. In SiloOS, an agent is none of that. An agent is a folder containing plain text files.

This is the **Markdown Operating System**—a radically simple architectural pattern where agents run on markdown files, not frameworks. These files aren't documentation. They're the actual operating instructions. Human-readable. Version-controllable. Auditable.

```
refund-agent/
  ├── main.py           # Entry point
  ├── tools.py          # Python tools
  ├── instructions.md   # What to do
  ├── escalation.md     # When to escalate
  └── templates/        # Email templates
```

That's it. That's an agent. A folder. Simple to understand, easy to deploy, inspectable by humans. The entire behavior of a customer service agent lives in plain text files that anyone can read.

The Four Components

Markdown OS rests on four pillars: **folders**, **markdown**, **Python**, and **scheduling**.

Folders are agent workspaces. Each agent has its own folder—a self-contained unit. Deploy by uploading. Update by replacing.

Markdown contains human-readable instructions. Policies, escalation rules, procedures. Not documentation—actual instructions the agent executes.

Python handles efficiency. Data-heavy operations that shouldn't go through the LLM. LLMs excel at reasoning but fail at math—Python handles calculations and API calls.

Why Markdown Works

- ✓ **Human-Readable:** Anyone can review policies
- ✓ **Version Control:** Git tracks all changes
- ✓ **LLM-Native:** Natural language instructions
- ✓ **Portable:** No framework lock-in

Scheduling provides automation. When and how agents run. The router decides when to wake them, what task to assign, when to shut down.

Token Efficiency: 4× Better

Model Context Protocol (MCP) sends tool descriptions in every request—thousands of tokens just describing what tools are available. For 20 tools, you burn 5,000-8,000 tokens per request before processing the customer's question.

Markdown OS loads instructions once. Tools are called directly, not described in context. Result: 4× more token-efficient. At 100,000 requests/day, that's 600M fewer tokens—roughly $1,200/day savings on Claude 4.5 pricing.

Deployment: Small, Atomic, Shippable

The folder is the deployment unit. Want to update the refund agent's escalation policy? Edit `escalation.md`. Commit. Deploy. Done. No months-long sprints. No coordination across teams. No deployment windows.

THE ROUTER AS KERNEL

The security foundation that makes safe AI operations possible

In any operating system, there's a critical principle: processes don't talk directly to hardware. They go through the kernel. The kernel manages resources, enforces permissions, and ensures no process can do something it shouldn't. In SiloOS, the router plays exactly the same role—it's the kernel that makes safe AI operations possible.

When you first think about multi-agent systems, the intuitive design is peer-to-peer: agents talking to other agents, collaborating directly. It sounds elegant. It feels like how humans work together.

It's also a security nightmare.

The Kernel Principle

Processes don't talk directly to hardware—they go through the kernel. In SiloOS, agents don't talk directly to each other—they route through the router.

Why Not Agent-to-Agent?

The temptation to build agent-to-agent communication is strong. Modern frameworks encourage it. But here's what actually happens: routing logic gets distributed everywhere, security boundaries become fuzzy, debugging becomes impossible, and monolithic codebases emerge.

The Router's Six Responsibilities

1. Task Receipt
Single entry point for all tasks

2. Agent Selection
Determines which agent handles it

3. Key Minting
Creates task keys for interaction

4. Dispatch
Sends task + keys to agent

5. Result Handling
Logs outcomes, manages escalations

6. Logging Everything
Complete audit trail

SiloOS makes a different choice: agents don't communicate directly. They route through the kernel. This centralization isn't a bottleneck—it's the source of SiloOS's security guarantees.

The Escalation Pattern

Agent can't handle a $700 refund (limit: $500)?

It doesn't call the manager directly.

It hands back to the router: "I can't do this.
Reason: amount exceeds authority.
Recommend: approve."

Router then routes to manager with fresh task keys. Clean, auditable, secure.

Single Point of Trust

By making the router the only trusted component, we make the entire system more secure. The router is the only component that can mint keys, route tasks, authorize escalations, and write to the audit log. Everything else—all the agents—is untrusted by design.

This means you write the router once, audit it carefully, lock it down. Then you can iterate on agents rapidly because they're operating in a security sandbox that doesn't depend on trusting them.

Keeping Agents Atomic

Perhaps the most underrated benefit: the router pattern keeps agents simple. When agents route through the kernel, they don't need to know about each other. They don't need complex multi-agent protocols. They just need to know: "If I can't handle this, return to router."

This is the antidote to framework bloat. With router-based orchestration, agents stay atomic—small, independently deployable units. Update one agent folder, push to git, deploy. No coordination needed. No other agents affected.

The router isn't overhead—it's the foundation that makes safe, scalable multi-agent operations possible. Centralized control, distributed execution.

ISOLATION

Containers, jails, and sandboxes—the kernel-level enforcement that makes agent containment real

Defense in Depth

Layer 1: JWT keys validate capabilities

Layer 2: Tokenization hides real data

Layer 3: Proxy enforces access control

Layer 4: Container isolation prevents escape

Layer 5: Network rules allow only proxy

If one layer fails, four others protect you.

The padded cell isn't just a metaphor. In SiloOS, every agent runs in genuine technical isolation—layers of operating system primitives that enforce containment. Linux jails, Docker containers, seccomp profiles, network namespaces: these are the bars on the cell door.

We've established that agents are untrusted. We've given them capabilities and scoped access. We've tokenized the data. But that's *software* enforcement. What happens if the agent finds a bug in your proxy?

Technical isolation is the answer. We run the agent in a restricted environment where it physically *cannot* access resources outside its cell—even if it tries. Even if our software has bugs.

> *"In our design, the AI literally cannot do X without permission—the identity system won't let it."*
>
> — Microsoft Engineering, "Zero-Trust Agents"

Not "shouldn't." Not "is discouraged from." **Cannot.** The architecture makes it impossible.

The Gold Standard: Docker + gVisor

Docker provides namespace isolation. Each container sees its own process tree, its own network stack, its own filesystem. From inside the container, the agent can't see other containers, can't see the host's processes, can't see the host's files. It's in a bubble.

gVisor adds a user-space kernel layer. Traditional containers share the host kernel—if the agent finds a kernel exploit, it can break out. gVisor intercepts all system calls and runs them through a sandboxed kernel implementation. Even kernel exploits can't escape.

Container Security

- ✓ **Drop all capabilities** — Agent has no privileged operations
- ✓ **Apply seccomp profile** — Whitelist only essential system calls
- ✓ **Read-only filesystem** — Agent can't modify code or persist files
- ✓ **Network isolation** — Only proxy connection allowed
- ✓ **Resource limits** — CPU, memory, disk quotas enforced

The Threat Model

Assume the agent is actively hostile. It might try to:

- × Access files it shouldn't (customer database, encryption keys)
- × Make network connections (exfiltrate data to external servers)
- × Escalate privileges (become root, install backdoors)
- × Consume unbounded resources (denial of service)

Technical isolation makes all of these impossible—not through the agent's good behavior, but through kernel-level enforcement.

Each agent invocation gets a *fresh* container. The filesystem is ephemeral. Any files written to `/tmp` vanish when the container terminates. The agent can't leave breadcrumbs, accumulate state, or persist malware. Every task starts from a known-good baseline.

PLUG IN A HUMAN

When the architecture proves itself: humans as fallback agents

This is the moment when you know the architecture is right. Your refund agent has been running smoothly for weeks, handling hundreds of customer cases per day. Then something changes. The agent starts approving refunds it shouldn't. You need to take it offline immediately for debugging.

The Inversion

Traditional: AI assists humans.

SiloOS: Humans assist AI.

The human is the fallback when AI needs maintenance or encounters edge cases.

In a traditional system, this would be crisis mode. Workflows break. Customers get errors. Engineers scramble. But in SiloOS, you simply mark the agent offline. The router sees it's unavailable and routes new tasks to a human operator instead. No crisis. No downtime. Just a channel switch.

The human uses the same interface, same tools, same security model. They see tokenized customer context— `[NAME_1]`, `[EMAIL_1]` —the conversation history, available tools from `tools.py`, instructions from `instructions.md`, and policy limits. They click "Validate Phone Number" (a tool the agent would have called programmatically), enter the amount, process the refund. Done.

Human as Agent

In SiloOS, a human operating through the interface **is** an agent:

- → Has base keys (job authority)
- → Receives task keys (access to this case)
- → Uses tools (via UI clicks)
- → Follows instructions (markdown)
- → All actions logged and auditable

This is what we mean by "plug in a human." The system doesn't care whether it's routing to Python code or a human operator—the boundaries are identical. When your architecture treats humans and AI as equivalent participants constrained by the same rules, you've achieved something rare: a security model that doesn't depend on *who* is executing, only on *what they're allowed to do*.

The use cases are immediate: **Maintenance windows** (update the agent without service interruption). **Testing** (route 1-in-100 cases to humans for comparison). **Edge cases** (fraud suspicion, regulatory exceptions). **Training data** (human handling reveals exactly where agents fail).

The Bottom Line

Remember: **95% of AI pilots stall before reaching production.** Not because AI isn't capable. Because we've been trying to solve an *architecture problem* with *alignment techniques*.

SiloOS answers with concrete patterns: base keys for capability, task keys for scope, tokenization for privacy, stateless execution for safety, router as kernel for orchestration, technical isolation for containment, everything logged for auditability.

Stop trying to trust AI. Build the cell instead.