Nick Clark · Published April 25, 2026 · PDF

How to Give an On-Device AI Agent Tools It Can Use Safely

You want a local AI agent that can call real tools (models, classifiers, retrievers) without letting those tools quietly do things your policy never approved. This guide describes an architecture for exactly that: tools as governed, managed assets the agent owns, with dispatch conditioned on the agent's state and every outcome recorded. It is an architecture disclosed in U.S. Provisional Application No. 64/070,239 (not a shipping library), and its home inventive step is the Agent-Resident Execution Substrate inventive step.

What You Are Building

You are building a personal or edge AI agent that runs on a device you control, and you want to hand it a set of tools it can actually use: a language model, a code-generation model, a speech recognizer, an image classifier, a retriever. The hard part is not wiring the calls. The hard part is safety. Once an agent can invoke tools, it can also invoke the wrong tool, on the wrong input, at the wrong time, and take an action nobody sanctioned. If those tools are updated or retrained, you also need to know that the update was authorized and be able to reconstruct what happened afterward.

The goal here is an architecture where every tool the agent can reach is a governed, managed asset: it carries its own model artifact, its own interface specification, and its own governance scope. The agent decides which tool to dispatch to based on its own

current state and policy, and every dispatch and every lifecycle change is written to an append-only record. This guide teaches the design so you can build it yourself. It is an architecture disclosed in a patent filing, not a package you download.

Why the Obvious Approaches Fall Short

The common ways of giving an agent tools each work, and each leaves a structural gap for on-device safety.

Tool-using agent frameworks let an agent invoke models and services through an orchestration layer. In these frameworks the agent is typically an ephemeral session or a stateless dispatch function that calls tools according to a declared workflow. The tools are external services the agent calls, not assets the agent owns and governs. There is no persistent authority deciding, per call, whether a tool is admissible for a request under a policy, and no accumulated record of tool outcomes that survives the session.

On-device inference frameworks give you a local model registry plus load, unload, and route utilities. That is genuinely useful, but these frameworks operate as request-routing infrastructure. They hold no persistent identity or state beyond your configuration, keep no lineage of what each model was asked and how it did, and enforce no governance policy over model lifecycle operations like installation or retraining.

Retrieval-augmented generation is often reached for when the goal is making the agent reflect your context safely. It supplements a base model with similarity search over your documents and injects fragments into the prompt at query time. It does not govern what the model may do, and it consults your corpus at every query rather than internalizing it.

None of these is wrong. The gap they share is that the tool is not a first-class governed thing the agent is accountable for. There is no single persistent authority on the device that owns the tools, conditions each dispatch on its own state and policy, and records the result so the history is reconstructible. That authority is what the architecture below supplies.

The Architecture

The disclosed approach makes the agent a persistent execution substrate on the device, and makes every tool a managed asset subordinate to it. Six pieces do the work.

The agent as persistent substrate. A semantic agent runs as a persistent computational entity for the lifetime of the device. It carries four persistent fields: a persistent identity field, a cognitive state field, an append-only lineage field, and a governance policy field. Per the spec, the agent's identity is not dependent on any specific model artifact and is preserved across replacement, retraining, or removal of any subordinate model. In the disclosed embodiment the agent is the sole authority over its own four fields; subordinate components may change state in their own domains but do not modify the agent's persistent fields except by appending to the lineage field.

The managed inference tool registry. This is a device-local data structure recording the tools currently installed. Each managed inference endpoint comprises three things: a model artifact (parameters plus architecture sufficient to run it), an interface specification (input format, output format, dispatch protocol), and an associated governance scope (the policy objects governing that endpoint's installation, retraining, replacement, archival, and removal). Endpoints of different types and sizes can be co-resident: general-purpose language models, task-specific fine-tuned models, image classifiers, speech recognizers, embedding models, retrieval models, and personal corpus models. The spec also describes a per-endpoint capability declaration recorded in the registry, signed by the endpoint's publisher and verified at installation

and at each lifecycle operation. It enumerates admissible input modalities, task categories, input and output formats, a latency profile, a resource envelope, a governance compatibility tag set naming which policy categories may govern the endpoint, and a capability confidence value.

Cognitive-state-conditioned dispatch. The agent-to-tool dispatcher routes each inference request, conditioned on four inputs: the current cognitive state field value, the request's input characteristics, the applicable policy objects, and the per-endpoint capability declarations. In the disclosed embodiment it determines the request's input modality and task category, matches those against capability declarations to find candidates, then consults cognitive domain fields (the spec names an execution confidence domain, a normative integrity domain, and a capability awareness domain) to derive confidence, normative, and capability thresholds. Candidates falling below those thresholds are excluded; among the rest, a selection function prefers endpoints with higher historical outcome quality recorded in the lineage field for similar requests. The dispatcher can also fan out to several endpoints and aggregate (majority vote, weighted average, and others), or run endpoints serially so one endpoint's output feeds the next.

Governed tool lifecycle. A tool lifecycle controller performs installation, retraining, replacement, archival, removal, quiescence, and resumption. The spec gives a state machine: uninstalled, installed-inactive, active, retraining, updated-active, archived, removed, with defined transitions. The safety-critical property is that each operation is initiated by the agent on a trigger event, evaluated against the governance policy field for admissibility, executed, and recorded in the lineage field. Retraining and substitution happen in a staging area distinct from the active registry, and the updated artifact is promoted to active only on successful completion and successful policy validation; if validation fails, the substitution rolls back and the failure and its cause are recorded.

Lineage recording. For each dispatched request the lineage controller appends a record with an input descriptor, an endpoint identifier, an output descriptor, a timestamp, and a downstream-outcome reference. That downstream reference points at how the output was later evaluated: user acceptance, user revision, downstream execution success or failure, and integrity-signal feedback from the cognitive domain fields. The lineage field is append-only under continuity proofs, so no prior record can be modified or deleted without producing a detectable continuity break; the spec describes cryptographic chaining where each record references its predecessor. Lineage is partitioned into per-endpoint sub-lineages, which is what the dispatcher's selection function reads to judge outcome quality.

The governance policy field. Policy objects are cryptographically signed, versioned, and machine-evaluable. Each lifecycle operation, dispatch, ingestion, and retraining event is evaluated against the applicable policy objects, and the policy version under which the operation was admitted is recorded, so you can later verify an operation was admissible under the policy in effect at the time even if the policy has since changed. Where multiple policy objects apply to one operation, the operation is admissible only if it satisfies the conjunction of all of them, except where an explicit override authority is declared.

Two further properties are worth naming because they are what make the tools safe rather than merely managed. First, the continuity guarantee: the agent's identity, cognitive state, and lineage are not modified by any lifecycle operation on a tool, by adding or removing a tool, or by a policy update, except by appending to the lineage field. You can replace every tool in the registry and the agent is structurally unchanged. Second, the privacy invariant (described in the spec for the broader substrate): lineage records, model artifacts, and corpus contents are not transmitted off the device except under an explicit disclosure policy object, with each disclosure recorded as a lineage event.

How to Approach the Build

A practical order for implementing this yourself.

1. Stand up the persistent agent and its four fields first. Before any tool exists, give the agent a persistent identity field, a cognitive state field, an append-only lineage field, and a governance policy field, all persisted to device storage and reloaded on restart. Everything else is subordinate to this. If you skip it and start with a tool router, you have rebuilt an on-device inference framework, not the substrate.

2. Define the managed inference endpoint record. Make the tool a structured record, not a function pointer. Illustrative sketch, faithful to the spec's fields (you implement the types):

```
endpoint = {
  model_artifact,          # parameters + architecture
  interface_spec,          # input format, output format, dispatch protocol
  governance_scope,        # policy objects governing this endpoint's lifecy
  capability_declaration   # modalities, task categories, latency profile,
                           # resource envelope, governance tags,
                           # capability confidence; publisher-signed
}
```

Verify the publisher-signed capability declaration at installation and at every lifecycle operation, as the spec directs.

3. Build the dispatcher as a filter-then-select function. Do not let it call a tool by name. Have it (a) derive input modality and task category from the request, (b) match those to capability declarations for candidates, (c) read the cognitive domain fields to get confidence, normative, and capability thresholds, (d) drop candidates below threshold, (e) among survivors, prefer higher historical outcome quality from the per-endpoint sub-lineage. Illustrative only:

```
def dispatch(request, agent):
    candidates = registry.match(request.modality, request.task)
    t = agent.cognitive_state.thresholds()
    admissible = [e for e in candidates
                  if e.confidence >= t.confidence
                  and e.normative >= t.normative
                  and e.capability >= t.capability
                  and policy.admits(request, e)]
    chosen = select_by_outcome_quality(admissible, request, agent.lineage)
    record = run_and_record(chosen, request, agent.lineage)
    return record.output
```

4. Route every lifecycle change through policy evaluation and staging. Install, retrain, replace, archive, and remove must each be evaluated against the governance policy field, executed in a staging area for retrain/replace, promoted only on successful policy validation, and rolled back with a recorded cause on failure. Record the admitting policy version.

5. Make lineage append-only and chained. Each record references its predecessor; a modified or deleted record breaks the chain detectably. Partition by endpoint so the dispatcher's selection function has a per-tool outcome history to read. Capture the downstream-outcome reference (accepted, revised, executed, failed) rather than only the raw output, because that is the training signal.

6. Add the continuity check and the disclosure boundary. Chain a continuity hash over lineage events so a break triggers escalation under policy. Gate all off-device transmission behind an explicit disclosure policy object and record each disclosure. These are what let you replace tools freely and still trust the agent, and what keep tool state on the device.

Sizing and tradeoffs the spec is explicit about: model artifacts are bounded to fit local memory, storage, and compute; adapter-based endpoints let a shared base model carry per-endpoint adapter weights to shrink the registry footprint; and foreground inference should be prioritized over background retraining and ingestion so user requests are not delayed by deferrable maintenance.

What This Does Not Give You

This is an architecture, not a drop-in library. There is no package to install and no SDK behind this guide. You implement the agent fields, the registry record, the dispatcher, the lifecycle controller, the lineage store, and the policy evaluator yourself, against your own runtime and hardware.

It is disclosed in a patent filing. It has not been benchmarked or productized here, and this guide states no latency, accuracy, or throughput numbers, because the spec states none and inventing them would be dishonest. The continuity guarantee, the staged atomic substitution, the append-only lineage integrity, and the policy conjunction are properties of the design as disclosed; whether your build achieves them depends on your implementation.

The approach targets on-device and edge deployments with bounded local resources and a governing user or operator policy. It is not aimed at multi-tenant cloud inference. It does not, by itself, make any individual model more accurate; it governs which tool is used, when, and under what policy, and records what happened. The cryptographic and identity primitives the spec references (signed policy objects, hardware-anchored identity, continuity proofs) are named as suitable mechanisms, not supplied as code here.

Disclosure Scope

The architecture described in this guide, including the agent-resident execution substrate, the managed inference tool registry, cognitive-state-conditioned dispatch, governed tool lifecycle operations, and lineage-recorded outcomes, is disclosed in U.S. Provisional Application No. 64/070,239. Its home inventive step is the Agent-Resident Execution Substrate inventive step. This guide is educational: it teaches an approach a developer can implement independently. It is not a warranty, not an offer of software, and not a representation that any product implementing this architecture exists or performs to any stated measure. Every mechanism described here traces to that filing; where the filing is silent, this guide makes no claim.

Agent-Resident Execution Substrate (/agent-resident-execution-substrate)

All 40 steps → (/inventive-steps)

Persistent execution environment carried by the agent, not the host — identity, state, and lineage across power cycles, devices, and upgrades.

Provisional application

PRIMARY TECHNICAL DISCLOSURE

- Agent-Resident Execution Substrate, Articles (/articles/agent-resident-execution-substrate)

SECONDARY TECHNICAL

- Persistent Semantic Agent (/articles/agent-resident-execution-substrate/persistent-semantic-agent)
- Managed Inference Tool Registry (/articles/agent-resident-execution-substrate/managed-inference-tool-registry)
- Agent-to-Tool Dispatcher (/articles/agent-resident-execution-substrate/agent-to-tool-dispatcher)
- Lineage-Derived Training Signal (/articles/agent-resident-execution-substrate/lineage-derived-training-signal)
- Identity Preservation Across Upgrades (/articles/agent-resident-execution-substrate/identity-preservation-across-upgrades)

- Cognitive State-Conditioned Dispatch (/articles/agent-resident-execution-substrate/cognitive-state-conditioned-dispatch)
- Governed Tool Lifecycle (/articles/agent-resident-execution-substrate/governed-tool-lifecycle)
- Continuity-Proof Lineage (/articles/agent-resident-execution-substrate/continuity-proof-lineage)
- Substrate Runtime Continuity (/articles/agent-resident-execution-substrate/substrate-runtime-continuity)
- Personal Corpus Model Training (/articles/agent-resident-execution-substrate/personal-corpus-model-training)
- Heterogeneous Inference Endpoints (/articles/agent-resident-execution-substrate/heterogeneous-inference-endpoints)
- Atomic Lifecycle Substitution (/articles/agent-resident-execution-substrate/atomic-lifecycle-substitution)
- Integrity Signal Feedback (/articles/agent-resident-execution-substrate/integrity-signal-feedback)
- Hardware-Bound Identity (/articles/agent-resident-execution-substrate/hardware-bound-identity)
- Cognitive State Append-Only Invariant (/articles/agent-resident-execution-substrate/cognitive-state-append-only-invariant)
- Counterparty Identity Records (/articles/agent-resident-execution-substrate/counterparty-identity-records)
- Privacy Egress-Controlled Disclosure (/articles/agent-resident-execution-substrate/privacy-egress-controlled-disclosure)
- Federated Cross-Device Agent Identity (/articles/agent-resident-execution-substrate/federated-cross-device-agent-identity)

APPLICATIONS · GENERAL

- Personal AI Agents That Survive Device Loss: One Continuous Identity and a Private Corpus Across Every Device (/articles/agent-resident-execution-substrate/personal-cross-device-agents)
- Enterprise Agent Fleets: Stable Agent Identity and Governed Tool Access Across Model Upgrades and Infrastructure Migration (/articles/agent-resident-execution-substrate/enterprise-agent-fleets)
- Audit-Grade Agent Identity for Regulated Finance and Healthcare: Continuity-Proof Lineage Across the Agent Lifecycle (/articles/agent-resident-execution-substrate/regulated-industry-agents)
- Edge and On-Device Agents: Hardware-Bound Identity Across Heterogeneous Inference Endpoints (/articles/agent-resident-execution-substrate/edge-and-on-device-agents)
- Agent-to-Agent Commerce With Counterparty Identity Records and Egress-Controlled Disclosure (/articles/agent-resident-execution-substrate/agent-to-agent-commerce)

- Governed Tool Lifecycles for Managed Inference-Provider Ecosystems: A Substrate Approach to Owning, Routing, and Retiring AI Tools (/articles/agent-resident-execution-substrate/managed-tool-ecosystems)
- Proving Unbroken Continuity in Long-Lived Autonomous Systems Across Substrate Migration and Atomic Model Substitution (/articles/agent-resident-execution-substrate/long-lived-autonomous-systems)
- Personal-Model Personalization: A User's Own Corpus-Internalized Model on the Agent-Resident Execution Substrate (/articles/agent-resident-execution-substrate/personal-model-personalization)
- On-Device Agent Identity for Robots and Autonomous Vehicles: An Auditable Substrate for Embodied Physical-World Agents (/articles/agent-resident-execution-substrate/embodied-physical-world-agents)

APPLICATIONS · SPECIFIC

- LangGraph Platform (LangChain) vs an agent-resident execution substrate: orchestration-graph state versus a portable, hardware-anchored agent runtime (/articles/agent-resident-execution-substrate/langgraph-platform)
- OpenAI AgentKit and the Assistants/Responses API vs agent-carried, hardware-anchored identity with governed tool lifecycle (/articles/agent-resident-execution-substrate/openai-agentkit)
- Microsoft Copilot Studio vs an agent-resident execution substrate: platform-hosted agent authoring versus portable, device-resident agent identity and continuity (/articles/agent-resident-execution-substrate/microsoft-copilot-studio)
- Google Vertex AI Agent Engine (managed runtime for deploying and scaling agents, with sessions/memory) vs an agent-carried, continuity-proofed identity substrate (/articles/agent-resident-execution-substrate/google-vertex-agent-engine)
- AWS Bedrock AgentCore (runtime, memory, identity, and gateway services for deploying agents at scale) vs an agent-resident execution substrate: where does the agent identity actually live? (/articles/agent-resident-execution-substrate/aws-bedrock-agentcore)
- Letta (formerly MemGPT) vs an append-only cognitive-state substrate: what a memory-management framework does not provide (/articles/agent-resident-execution-substrate/letta-memgpt)
- Cognition's Devin, an autonomous AI software-engineering agent vs a portable, continuity-proofed agent-resident runtime (/articles/agent-resident-execution-substrate/cognition-devin)
- Cloudflare Agents (Durable Objects) vs an agent-resident execution substrate: portable hardware-bound identity and continuity-proof lineage (/articles/agent-resident-execution-substrate/cloudflare-agents)
- Ollama alternative: from local model runner to a governed agent-resident substrate (/articles/agent-resident-execution-substrate/ollama)

- Apple Intelligence (on-device foundation models, Private Cloud Compute) vs a persistent agent-resident substrate: who owns identity, lineage, and the model? (/articles/agent-resident-execution-substrate/apple-intelligence)

Agent-Resident Execution Substrate overview → (/agent-resident-execution-substrate)