Nick Clark · Published July 2, 2026 · PDF

How to Build a Personal AI That Runs on Your Own Device

Cloud assistants treat you as an anonymous row in a shared model: you ship your context up on every request, and it is discarded the moment the response comes back. This guide teaches a different architecture, one where a persistent agent lives on your own hardware, owns its inference models as governed local tools, and learns your body of work into weights that never leave the device unless you say so. The approach here is disclosed in U.S. Provisional Application No. 64/070,239 (it is an architecture, 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 AI that lives on hardware you control: a laptop, a desktop, a workstation, a phone, or an embedded device. It should do three things that a cloud chatbot does not. First, it should be a continuous entity, not a fresh session each time you open it, so that it remembers its own operational history across restarts, model swaps, and even software updates. Second, it should get better at your specific work, not by stuffing your files into a prompt window on every call, but by internalizing your terminology, structure, and prior output into the model itself. Third, it should keep your data on the device by default, with any transmission off the device being an explicit, audited decision rather than a silent background fact.

This is the search-intent problem behind "how do I build a personal AI that runs on my own device." It shows up for developers who want an assistant tuned to their own codebase and prose without sending everything to a third party, for professionals under confidentiality obligations, and for anyone who wants an assistant that accrues value over years instead of resetting every conversation.

The architecture below comes from a filed patent disclosure. It is a design you implement yourself against models and runtimes of your choosing. It is not a package you install, and nothing here is benchmarked or productized.

Why the Obvious Approaches Fall Short

There are several honest ways people attempt this today. Each is legitimate. The point is structural, not a criticism.

Cloud inference with per-request context. You call a hosted large language model and paste in whatever context you need. The model is shared across all users and undifferentiated by identity; it holds no persistent representation of your individual body of work, and the context window is discarded between requests. So you re-supply your context every single time, and the model never actually gets to know you.

Retrieval-augmented generation. You index your documents, do a similarity search at query time, and inject the retrieved chunks into the prompt. This helps, but the corpus is consulted at each query rather than internalized. Output quality rides on retrieval quality and chunk boundaries, and the base model's intrinsic representation of your corpus never improves.

Local inference frameworks. You run a model locally with a loader and a router to a local endpoint. This gives you privacy and offline capability, but these frameworks are request-routing infrastructure. They keep no persistent identity or state beyond

configuration, accumulate no history of inference outcomes, enforce no governance over model lifecycle, and preserve no continuous entity when you replace a model.

Manual fine-tuning services. You curate a dataset, kick off a training job, and deploy the artifact. Continuous, automated improvement from your ongoing work is not provided; the training lifecycle is decoupled from your authoring activity, so it goes stale the moment you stop babysitting it.

The gap they share: none provides a single persistent, identity-bearing thing on your device that owns its models as governed assets, records what happened, and uses that record to keep its models current with your actual work, while keeping the whole loop local. That missing thing is what you are building.

The Architecture

Everything below traces to U.S. Provisional Application No. 64/070,239.

A persistent semantic agent is the runtime authority of the device. Rather than an app that starts and stops, the substrate hosts a semantic agent that persists across the lifetime of the device. Inference endpoints, ingestion modules, and related computational assets are managed components subordinate to it. The agent comprises four persistent fields: a persistent identity field, a cognitive state field, an append-only lineage field, and a governance policy field. The agent's identity does not depend on any specific model; models can be replaced, retrained, or removed and the agent is unchanged. This is the continuity guarantee.

Models are managed tools in a local registry, not the agent itself. The managed inference tool registry is a device-local data structure recording each installed endpoint. Every endpoint carries a model artifact (parameters plus architecture), an interface specification (input format, output format, dispatch protocol), and a governance scope. Multiple endpoints of distinct types and sizes can be co-resident:

general models, task-specific fine-tunes, embedding models, and personal corpus models. The disclosure also describes adapter-based variants where a shared base model is specialized per endpoint with small adapter weights, cutting the registry's storage footprint.

A dispatcher routes each request, conditioned on state. The agent-to-tool dispatcher takes an inference request, reads its input modality and task category, matches those against each endpoint's capability declaration, and selects one or more endpoints. Selection is also conditioned on the cognitive state field and on which endpoints have higher historical outcome quality for similar inputs, as recorded in the lineage. The disclosure describes fanning one request out to several endpoints and aggregating (majority vote, weighted average, and similar), or chaining endpoints serially so one endpoint's output feeds the next.

The lineage field is the memory and the training signal. Every dispatch, outcome, lifecycle operation, ingestion event, policy change, and more is appended as a lineage record. It is append-only under a continuity proof: records can be chained cryptographically so that each references its predecessor, and no prior record can be altered without producing a detectable break. Each dispatch record carries an input descriptor, the endpoint identifier, an output descriptor, a timestamp, and a downstream-outcome reference, meaning whether you accepted the output, revised it, whether downstream execution succeeded or failed, and integrity-signal feedback. The disclosure makes a specific point here: because this training signal reflects real-world acceptance, revision, and success or failure rather than the model's own prior output, it carries information not generated by the model under training, which the disclosure frames as mitigating the distributional collapse seen when a model is trained on its own output.

The personal corpus model internalizes your work. This is the heart of the on-device learning loop. A personal corpus model is an endpoint whose parameters are fine-tuned against a corpus derived from artifacts you authored, curated, or designated.

Its behavior reflects your terminology, structural conventions, and prior outputs, and it does so without retrieval at inference time; your body of work is encoded in the weights rather than consulted as external context. The loop, per the disclosure, is closed: you author an artifact in a host application; the artifact is registered in the lineage; a corpus assembly module periodically selects admissible artifacts; a fine-tuning module applies a parameter-efficient update (low-rank adaptation, prefix tuning, prompt tuning, or continuous adapter training) sized to fit the device's local compute envelope and a policy-declared training window; a governed substitution promotes the updated artifact into the registry under the continuity guarantee; and the improved model then assists your next authoring, whose artifacts feed the next iteration.

Governance policy is the enforcement layer. The governance policy field holds cryptographically signed, versioned, machine-evaluable policy objects. Every lifecycle operation, dispatch, ingestion, and mutation is evaluated against the applicable policy, and the policy version under which it was admitted is recorded, so you can later verify an operation was admissible under the policy in force at the time. Policy objects are scoped to roles: user policy, system policy, corpus policy governing what may enter training, and more.

The privacy invariant keeps data on-device by default. The disclosure specifies that lineage records, model artifacts, training corpora, personal corpus model parameters, scope-local context, and counterparty records are not transmitted off the device except under an explicit disclosure policy object identifying a recipient, a permitted scope, an authorization attestation, a retention requirement, and a revocation mechanism. Every off-device disclosure is itself recorded as a lineage event, so the full history of what left the device is auditable. Enforcement mechanisms the disclosure names include a substrate-runtime egress filter over outbound traffic, per-component isolation so subordinate components cannot transmit on their own, and hardware-anchored attestation that the runtime mediating disclosure is untampered. The invariant is operative regardless of connectivity, and inference done entirely on-device is not a disclosure at all.

Cloud-burst forwarding is the escape hatch, governed. When a local endpoint lacks the capability or capacity for a request, a cloud-burst subsystem can forward it to a remote endpoint, but only under a cloud-burst policy object naming admissible remote endpoints, request categories, disclosure scopes, encryption requirements, cost limits, and prohibited categories. Before forwarding, the disclosure describes an admissibility test: a capability test (can any local endpoint do this), a capacity test (is there local compute for the latency budget), a disclosure test (are these inputs allowed off-device), and a cost test. Every forward is a lineage event and is treated as an off-device disclosure. A deferred mode queues requests for the next connectivity event; a confidential-execution mode encrypts payloads to a remote trusted execution environment.

Hardware-anchored identity binds the agent to the device. The persistent identity field can be cryptographically bound to a hardware security element (a secure enclave, TPM, hardware security module, or embedded secure element) whose private key material is not extractable. The binding is derived at first instantiation and re-verifiable at any later event. A hardware-bound identity is not transferable to another device except under a governed migration operation with attestations from both devices, and the disclosure describes quarantining agent execution if the hardware fails attestation.

Scopes partition one identity into many contexts. The cognitive state field can hold named scope records, each with its own corpus policy, tool subset, and lineage partition, so a single agent identity maintains independent professional, personal, and project contexts. An active-scope indicator, set by you or inferred from context, decides which scope a request runs against.

How to Approach the Build

You are implementing an architecture, so the work is defining the fields, the registry, and the policy-checked operations, then wiring a local model into them. A reasonable order:

1. Define the agent's four persistent fields and persist them. Model the identity field, cognitive state field, lineage field, and governance policy field as durable, device-local structures. The lineage field is the one to get right first: make it append-only with each record chaining a reference to its predecessor, so tampering is detectable. An illustrative record shape, faithful to the disclosure and clearly just a sketch:

```
LineageRecord {
  prev_hash        // reference to the previous record
  event_type       // dispatch | outcome | lifecycle | ingestion | policy |
  input_descriptor
  endpoint_id
  output_descriptor
  outcome_ref      // accepted | revised | exec_success | exec_failure | int
  scope_id
  policy_version
  timestamp
}
```

2. Build the tool registry and a capability declaration per endpoint. Store, for each endpoint, the model artifact reference, interface spec, and governance scope, plus a capability declaration enumerating admissible modalities, task categories, input and output formats, a latency profile, size, and resource envelope. The dispatcher will read these to route.

3. Implement the dispatcher as a policy-checked router. Given a request, determine modality and task category, filter to endpoints whose capability declaration matches, evaluate applicable policy, and select. Start with single-endpoint routing; add aggregation and serial chaining later. Record every dispatch and its outcome to the lineage.

4. Wire in a local base model as your first managed endpoint. Use any on-device inference runtime you like to execute a bounded-size model locally, and register it in the registry as an endpoint rather than calling it directly. The agent should reach models only through the dispatcher, never around it.

5. Stand up the tool lifecycle controller with staging and rollback. Install, activate, retrain, replace, archive, and remove endpoints as governed operations recorded in the lineage. Do retraining and substitution in a staging area distinct from the active registry, and promote a new artifact only after it passes policy validation; on failure, roll back to the prior artifact and record the cause. This is what makes model swaps safe without disturbing the agent identity.

6. Close the personal corpus loop. Register authored artifacts to the lineage. Build a corpus assembly step that selects lineage artifacts admissible under a corpus policy, applies any redaction rules, and optionally filters by scope. Apply a parameter-efficient fine-tune sized to your device's compute and a bounded training window, then run the update through the same staged substitution from step 5. Schedule it for idle or low-power windows, and prioritize foreground inference over background retraining so user requests are never blocked by maintenance work.

7. Enforce the privacy invariant at the boundary. Put every outbound path behind an egress filter that checks a disclosure policy object before anything leaves, and record each disclosure to the lineage. Make subordinate components incapable of transmitting on their own. Treat cloud-burst forwarding as a disclosure that must pass the capability, capacity, disclosure, and cost tests before it fires.

8. Add scopes, hardware binding, and resource governance as you harden. Introduce named scope records once the single-scope loop works. If your device has a hardware security element, derive the identity binding from it. Add per-tool resource budgets (memory, storage, compute, power) with eviction and quiescence so the substrate degrades gracefully under thermal or battery pressure.

What This Does Not Give You

This is an architecture disclosed in a patent filing, not a drop-in library. There is no SDK to install and no repository that ships this for you; you implement each component yourself against models and runtimes you choose. The pseudocode above is illustrative, not working code.

Nothing here is benchmarked or production-proven. The disclosure describes structural properties, for example that lineage-derived training signal carries real-world outcome information rather than recycled model output, and it frames that as mitigating distributional collapse. It does not state performance numbers, training times, model sizes, or accuracy figures, and neither should you infer them. Whether an on-device fine-tune actually improves your outputs depends entirely on your model, your data, and your compute, none of which this design specifies.

The privacy invariant is an architectural enforcement you must actually build; describing a disclosure policy does not encrypt anything by itself. On-device fine-tuning is bounded by real hardware, so a small device may only support aggressively quantized endpoints and infrequent retraining. And this architecture is aimed at personal and edge deployments where a single user or operator governs the device; it is not a design for multi-tenant cloud inference, and it does not replace a hosted model when you genuinely need frontier-scale capability, which is exactly the case the governed cloud-burst path exists to handle.

Disclosure Scope

The architecture described in this guide is disclosed in U.S. Provisional Application No. 64/070,239, titled for an agent-resident execution substrate with a governed inference tool registry and lineage-derived personal corpus model training. This guide is educational: it teaches how to approach building a personal, on-device AI using the disclosed architecture, and it traces its technical claims to that filing. It is not a warranty, not an offer of software, and not a representation that any implementation is complete, benchmarked, or fit for a particular purpose. You are responsible for your own implementation, and references to third-party technologies such as trusted platform modules, secure enclaves, and low-rank adaptation are provided for context and described as they are commonly understood.

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)