Nick Clark · Published April 25, 2026 · PDF

How to Fine-Tune a Personal Model on Your Own Writing Without Sending It to the Cloud

You want a model that writes and reasons like you do, trained on your own artifacts, but you do not want to ship your corpus to a hosted service to get there. This guide describes an architectural approach to building that: a device-local model whose weights internalize your body of work, kept current automatically as you keep writing. It is an architecture disclosed in U.S. Provisional Application No. 64/070,239, not a shipping library. The approach centers on the home inventive step named there: the Agent-Resident Execution Substrate inventive step.

What You Are Building

You are building a model that produces output reflective of your own domain knowledge, terminology, structural conventions, and prior outputs, and that runs and trains entirely on hardware you control. The searcher's problem is specific: not "give me a model that answers questions," but "give me a model that has internalized my writing, without my writing leaving the machine."

The disclosed approach calls the trained artifact a personal corpus model. Its defining property, per the filed specification, is that its behavior is determined by its weight parameters rather than by retrieval at inference time. Your accumulated body of work is

internalized in the parameter values, not consulted as external context on each query. That distinction is the whole point of the design, and it is what separates this from the retrieval approaches most developers reach for first.

This is for a developer or technical writer who owns a growing corpus of artifacts (prose, code, publications) and wants a model conditioned on that corpus, held to a hard constraint that nothing about the corpus is transmitted off-device unless explicitly authorized.

Why the Obvious Approaches Fall Short

There are three common ways people attempt this, and the specification's background section describes each accurately.

Retrieval-augmented generation is the most common. You compute embeddings over your documents, do similarity search at query time, and inject the retrieved fragments into a base model's prompt. This works, but it does not internalize anything. The base model's intrinsic representation of your corpus never improves; every query re-consults an index, and output quality is bound to retrieval quality, chunk-boundary effects, and the base model's ability to integrate what it was handed. Your corpus remains external to the model.

Hosted fine-tuning services can produce internalization, but they invert the privacy property you want. You curate a dataset, upload it, initiate training, and deploy the artifact. The corpus is handled by the service. The specification also notes a workflow gap: this lifecycle is decoupled from your authoring activity. It is a discrete, manual event, not a continuous reflection of your ongoing work.

On-device inference frameworks solve locality but not learning. They load and route local models; they are request-routing infrastructure. They keep no persistent state beyond configuration, accumulate no record of how inferences turned out, and drive no

retraining from your activity.

The structural gap common to all three: none maintains a persistent local entity that records what you author, governs what may enter training, retrains a local model from that record on a schedule, and holds the whole loop inside a device boundary. That gap is what the disclosed architecture targets.

The Architecture

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

A persistent local agent owns the loop. The substrate is organized around a semantic agent that persists as the execution substrate of the device. It holds four persistent fields: an identity field, a cognitive state field, an append-only lineage field, and a governance policy field. Model artifacts are not the agent; they are subordinate managed assets. The agent's identity is not dependent on any specific model and is preserved across replacement or retraining of any model it holds.

Authored artifacts are recorded in an append-only lineage. This is the closed loop described in the specification's personal corpus model section. When you author an artifact through a host application (a text editor, a code editor), that artifact is registered in the agent's lineage field under your governance policy. The lineage record encodes a content reference, the modality, an authoring timestamp, a scope identifier, and any declared admissibility metadata. The lineage is append-only under a continuity proof: prior records cannot be silently modified or deleted.

A governance policy decides what may train the model. Nothing enters the training corpus by default. A corpus policy object declares which artifacts are admissible, admissibility thresholds, and redaction or anonymization rules. This is the governed part of the loop, and it is what lets you keep, for example, drafts out of training while keeping finished work in.

Corpus assembly derives an incremental training set. A corpus assembly module periodically, or on a trigger, walks the lineage field and selects artifacts admissible under the corpus policy, filters for modality compatibility with the target model, applies declared redaction, and optionally filters by scope. The result is an incremental training set reflecting what you have authored since the last retraining event, not a full re-curation.

Fine-tuning is parameter-efficient and local. The fine-tuning module applies a parameter-efficient fine-tuning operation to the model's parameters. The specification names low-rank adaptation, prefix tuning, prompt tuning, and continuous adapter training as applicable techniques. The operation is deliberately configured to update a bounded subset of parameters so it stays feasible within the device's local compute envelope and completes within a policy-declared training window. The specification also describes the personal corpus model operating as a specialization layer over a frozen base model: the base supplies general capability, the personal layer bends output toward your body of work.

The updated model is swapped in under a governed, reversible operation. The new artifact is produced in a staging area distinct from the active registry and promoted to active only on successful completion and policy validation. If validation fails, the substitution is rolled back and the prior artifact stays active, with the failure recorded in lineage. This is the tool lifecycle state machine: an endpoint moves from active to retraining, then either substitutes to updated-active or rolls back to active.

The loop closes. The updated model serves your subsequent authoring, and the artifacts you produce with its help re-enter the lineage to feed the next assembly and fine-tuning cycle. The specification frames retraining frequency as a function of a policy-declared schedule, your authoring rate, available local compute, and outcome quality signals, with a policy-declared lower bound so the model does not drift too far from your current work.

The privacy invariant is what keeps it off the cloud. Separately from the training loop, the specification defines a privacy invariant: lineage records, model artifacts, training corpora, and personal corpus model parameters are not transmitted off-device except under an explicit disclosure policy object naming a recipient, a permitted scope, an authorization attestation, and a retention requirement. The specification describes enforcing this through mechanisms such as a substrate-runtime egress filter over outbound traffic and per-component isolation preventing subordinate components from transmitting on their own. Critically, the invariant is stated to be operative regardless of network connectivity, and inference performed entirely on-device is explicitly not an off-device disclosure.

Cloud-burst is the escape hatch, and it is governed. When a local model lacks capability or capacity, a cloud-burst forwarding subsystem may forward a request to a remote endpoint, but only after an admissibility test: a capability test, a capacity test, a disclosure test (are these inputs admissible off-device at all), and a cost test. Forwarded payloads are treated as off-device disclosure events and recorded in lineage. This is how the architecture stays honest: leaving the device is possible but never silent.

How to Approach the Build

You are implementing this yourself. The steps below are the order a developer would work in.

1. Stand up the persistent agent and its four fields. Before any training, you need a local process that persists across restarts and holds identity, cognitive state, an append-only lineage, and policy. The lineage is the spine of everything downstream, so build it first as an append-only log with a continuity check (each record referencing its predecessor).

2. Instrument authoring to write lineage records. Hook your editors so that each saved artifact appends a lineage record. Keep the record to references and metadata, faithful to the specification's fields:

```
# Illustrative only; faithful to the disclosed lineage record fields.
lineage.append(
    content_ref = store(artifact),      # reference, not necessarily inline
    modality    = "text/prose",
    authored_at = now(),
    scope       = "professional",
    admissibility = {"stage": "final"}, # your corpus policy reads this
)
```

3. Write the corpus policy before you train anything. Decide, as machine-evaluable rules, what is admissible: which scopes, which stages, what redaction. This is a gate, not an afterthought. If you skip it, you have built an ungoverned data pipeline, which is exactly the property you were trying to avoid.

4. Build corpus assembly as a query over lineage. Assembly reads the lineage, applies the corpus policy, filters by modality and scope, applies redaction, and emits an incremental set since the last retraining marker. Keep it incremental; the loop's whole economy depends on not re-curating from scratch.

5. Choose a parameter-efficient method and a base. Pick one of the methods the specification names (low-rank adaptation is the common starting point) and, if you follow the specialization-layer embodiment, freeze a base model and train only the personal layer. Size the base and the update so a training run fits your device's memory, compute, and thermal envelope and finishes inside your declared window.

6. Run fine-tuning in a staging area, validate, then substitute. Never train in place. Produce the artifact in staging, run your policy validation against it, and only then promote it to the active registry. Wire the rollback path so a failed validation leaves the prior model active. Record both outcomes in lineage.

7. Set the schedule and its lower bound. Trigger retraining on your authoring rate and idle/power windows, but enforce a policy-declared minimum frequency so the model cannot silently drift behind your current work.

8. Add the egress filter last, and make it fail closed. Implement the privacy invariant as an egress control that denies any off-device transmission of corpus, parameters, or lineage absent a matching disclosure policy, and logs denials. Only after that exists should you consider cloud-burst, gating it behind the four-part admissibility test and recording every forward as a disclosure event.

An honest sequencing note: build steps 1 through 4 (agent, lineage, policy, assembly) and prove the loop records and gates correctly before you invest in the training method. The training is the part every framework already helps with; the governed, local, self-feeding loop is the part you are actually here to build.

What This Does Not Give You

This is an architecture, not a drop-in library. There is no package to install and nothing here "just works." You implement the agent, the lineage store, the policy evaluator, the assembly module, the fine-tuning integration, and the egress filter yourself.

The approach is disclosed in a patent filing. It is not a benchmarked or productized system, and the specification states no performance numbers, training-time figures, or quality metrics; do not expect any here, and do not infer them. Where the specification says a parameter-efficient method keeps training within a local envelope, that is a design intent you must validate on your own hardware, not a guarantee.

The design does not eliminate the base model's limitations. If you specialize over a frozen base, the personal layer bends output toward your corpus but inherits the base's general capabilities and gaps. Retrieval is not made obsolete for every task either; a small local corpus that changes faster than your retraining schedule may still be better served by consulting it directly for fast-moving facts.

Finally, the privacy invariant is only as strong as your implementation of it. The specification describes enforcement mechanisms (egress filtering, component isolation, hardware attestation), but a real egress filter that fails open, or a component allowed to transmit outside it, breaks the property. The architecture tells you where the boundary belongs; holding the boundary is your engineering responsibility.

Disclosure Scope

The approach described in this guide is disclosed in U.S. Provisional Application No. 64/070,239, which describes an agent-resident execution substrate with a governed inference tool registry and lineage-derived personal corpus model training. This guide is educational. It explains an architectural approach so a skilled developer can build it themselves; it is not a warranty, a specification of a shipping product, or an offer of software. Any real technologies referenced for context are described as they generally work and are not represented as part of the disclosed invention. Nothing here should be read as a claim that a benchmarked or production-proven implementation exists.

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)