MachineFabric & CapDAG

A Local-First AI Orchestration Platform and the Typed Plugin Protocol That Powers It

machinefabric.com | capdag.com | github.com/machinefabric

April 2026

Abstract

MachineFabric is a local-first AI document processing application for macOS. Users select a new "Transmute" option in the finder context menu on their files (PDFs, images, audio, text), and MachineFabric routes them through a graph of AI plugins ("cartridges") that extract, transcribe, embed, summarize, classify or otherwise digest the content. All processing happens on the user's machine. Nothing is sent to the cloud.

MachineFabric is built on **CapDAG**, an open protocol for typed capability dispatch. Each plugin carries a structured identifier describing what it accepts, what it produces, and how it should be selected when multiple plugins overlap. Routing follows a formal dispatch rule based on standard function-subtyping semantics, which removes a class of routing ambiguities that informal plugin systems tend to leave undefined. The full formalism, wire protocol, and routing internals are in the technical appendix.

1. The Problem

A user in 2026 who wants to process a library of documents locally with AI runs into the same set of integration problems again and again:

- A PDF extractor (pdfplumber, PyMuPDF, PDFKit)
- An OCR model for scanned pages
- An embedding model (sentence-transformers, Candle, MLX)
- An LLM for summarization or Q&A (Ollama, LM Studio, MLX)
- Glue code connecting all of it

Each tool has its own CLI, API, data format, error handling, and assumptions. Wiring them together typically produces one-off Python scripts, fragile shell pipelines, or hand-built desktop apps that re-solve the same problem from scratch.

The available answers split along two unhelpful axes. Cloud-first orchestration frameworks (LangChain, LlamaIndex, MCP) assume network-first architecture and do not fit privacy-sensitive or offline workflows. Single-purpose local tools (Ollama, LM Studio, Rewind, DEVONthink) are strong in their lanes but do not compose: you cannot pipe Ollama's output into DEVONthink's index and then through a custom OCR model without writing code. What is missing is a **local-first composition layer** in which typed plugins are first-class on the user's machine.

2. What We Built

2.1. MachineFabric: the product

A native macOS application with a Swift user interface and a Rust execution engine. The user selects files in the Finder, right-clicks, and chooses "Transmute"; MachineFabric identifies each file's media type and routes it through the appropriate cartridges.

Four cartridges ship with the app: `pdfcartridge` (PDF text and image extraction), `txtcartridge` (text and markdown), `candlecartridge` (vector embeddings via the Candle Rust ML framework), and `mlxcartridge` (LLM inference via Apple's MLX). Third-party cartridges install through a standard path on disk, must be code-signed and Apple-notarized, and run in sandboxed XPC services. Each declares its capabilities in a JSON manifest stating what it accepts, what it produces, and what arguments it takes.

Example. Summarize each page of a PDF:

```
document.pdf  (media:pdf)
  → pdfcartridge   [cap:in="media:pdf";op=disbind;
                         out="media:textable;page"]
  → page text      (media:textable;page)
  → mlxcartridge   [cap:constrained;in="media:textable";language=en;
                         op=summarization;
                         out="media:generated-text;textable;record"]
  → summary        (media:generated-text;textable;record)
```

The user writes no glue code; MachineFabric resolves the graph from registered capabilities and executes it.

2.2. CapDAG: the protocol

CapDAG is the open specification for how capabilities are named, typed, and matched to requests. It is independent of MachineFabric: any runtime, in any language, can conform by implementing URN normalization, the dispatch predicate, and the Bifaci wire protocol.

A capability URN names an input media type, an output media type, and a set of non-directional tags (operation, model, format, and so on). When a request arrives, a single rule decides which providers are eligible and which is the best fit: standard function subtyping with contravariant inputs, covariant outputs, and refinement-matched tags. The point is not the math, which is textbook, but that this is the only rule. Routing stays inspectable and reproducible, and third-party cartridges get a stable contract. The grammar, dispatch predicate, and refinement rules are in Appendix A.

3. Architecture

MachineFabric runs as two processes communicating over gRPC. The **Engine (Rust)** is the dispatch and execution core: capability router, cartridge host, SQLite metadata store, and the executor that runs the DAG of cartridge invocations. The **App (Swift)** is the user surface: a list of "machines" (previously-run pipelines, each a DAG with recorded inputs, outputs, and execution traces), an

editor for the Machine Notation that drives them, a task interface for in-flight runs, a cartridge install manager, and the XPC bridge to sandboxed cartridges.

Cartridges are standalone executables in any language. They communicate with the engine over **Bifaci**, a framed stdin/stdout protocol with multiplexed streams, a handshake, identity verification, and keepalive. The routing topology and protocol details are in Appendix B.

Execution is unremarkable in a useful way: the engine builds a capability graph from registered cartridges at startup, and the planner finds a path through it when the user requests an operation. The graph is bounded, the same inputs always resolve to the same path, and the path is inspectable in the app. This is pathfinding, not general AI planning.

4. Why Now

Three shifts in the past eighteen months have made this the right moment:

1. **Apple Silicon and MLX** have made on-device inference genuinely usable. Llama-3-8B-class models run at conversational speed on consumer M-series Macs. Whisper transcribes in real time.
2. **Quantized open-weight models** (Llama, Mistral, Qwen, Phi) have closed enough of the quality gap with frontier cloud models to be useful for extraction, summarization, classification, and Q&A over personal data.
3. **Privacy pressure** (GDPR enforcement, AI-training lawsuits, enterprise data-residency requirements) has turned "your files never leave your machine" from a curiosity into a material selling point.

The capabilities are here. The composition layer that lets users wire them together without reinventing the plumbing each time is what is still missing.

5. Positioning

vs. Model Context Protocol (Anthropic). MCP is an emerging protocol gaining adoption quickly. It standardizes how LLM clients connect to external tools, resources, and prompts. MachineFabric targets a different surface: local file-processing pipelines where the unit of work is a typed media transformation (PDF $\rightarrow$ text, audio $\rightarrow$ transcript, text $\rightarrow$ embedding), not an agent-session tool call. CapDAG's type system is narrower and more formal because it supports automatic pipeline resolution without a planning LLM in the loop. The two are complementary; an MCP adapter that bridges cartridges and tools in both directions is a natural roadmap item.

vs. LangChain / LlamaIndex. Developer frameworks for LLM applications, agents, RAG, and data connectors. They can use local models, but they are libraries, not native end-user products, and the composition is something the developer writes and hosts. MachineFabric moves that composition into a macOS product surface with deterministic dispatch underneath.

vs. Ollama / LM Studio. Local model runtimes that serve inference. MachineFabric builds on that layer and operates above it: file intake, media extraction, routing, execution history, cartridge composition. They run models; MachineFabric composes model-backed and non-model cartridges into pipelines.

vs. DEVONthink / Rewind / Readwise. End-user knowledge-management and personal-memory products with built-in AI features. The difference is the extensibility model: they are vertically integrated, while MachineFabric exposes an open typed cartridge-routing ecosystem that third parties can extend.

vs. Spacedrive. The closest neighbor on the local-first axis. Spacedrive's center of gravity is file indexing, sync, search, and a personal data layer. MachineFabric's center of gravity is AI document transformation through typed dispatch.

vs. typed component and AI workflow projects. Gambit (typed LLM decks with Zod), **Burr** (typed state machines for AI applications), and **wasmCloud** (WebAssembly Interface Types for distributed components) occupy parts of the same typed-composition design space. They are frameworks and runtimes: developer surfaces. MachineFabric's combination of native end-user file orchestration, media-type routing, and a protocol-first dispatch model is what places it in a different quadrant.

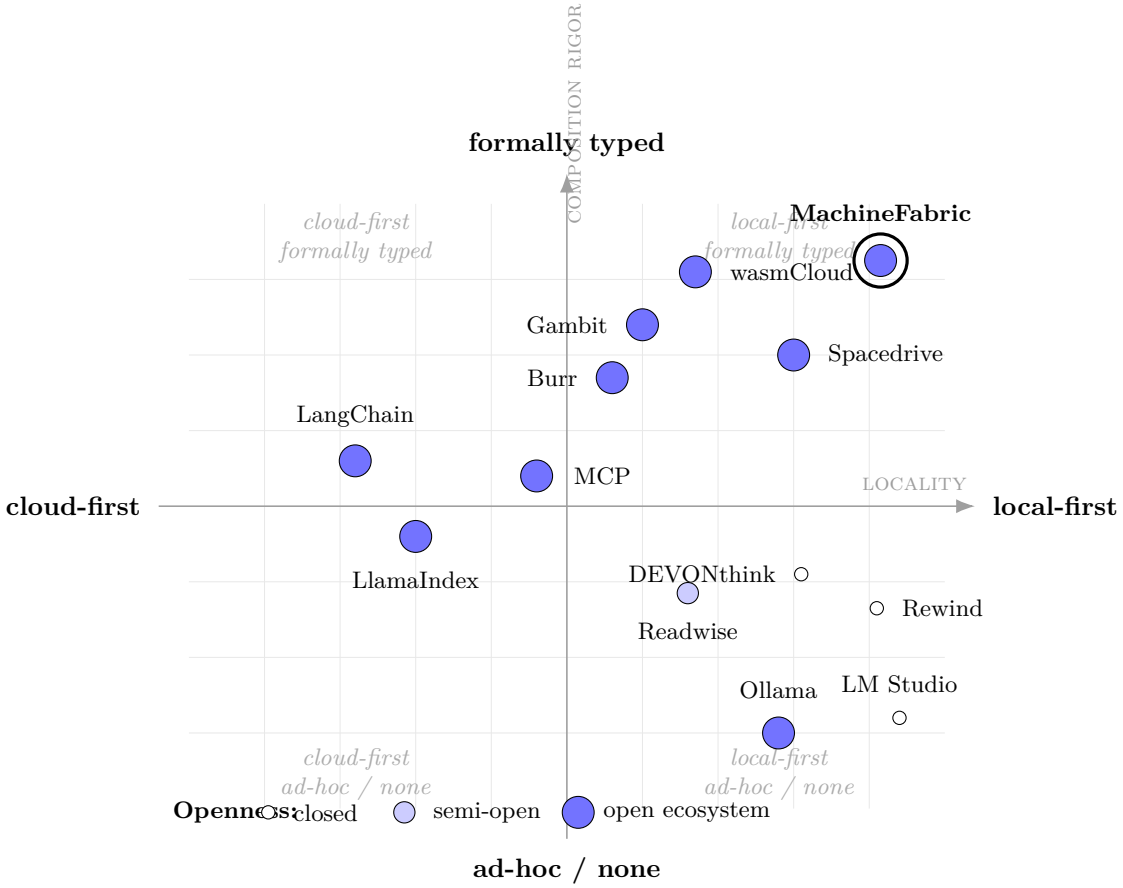

Figure 1: Positioning of MachineFabric against adjacent orchestration and local-AI projects. Axes cross at the center, dividing the plane by locality (cloud-first vs. local-first) and composition rigor (ad-hoc/informal vs. formally typed dispatch). Marker size and fill encode the third dimension, openness (closed product → open ecosystem). MachineFabric is distinguished by the combination of native end-user file orchestration, media-type routing, deterministic cartridge dispatch, and protocol-first extensibility.

6. Ecosystem Strategy

Long-run value depends on cartridges we do not write. Four principles:

Cartridges are first-class. Third-party developers can write them in any language. The Bifaci protocol is minimal, the manifest format is JSON, and distribution is gated by code signing and notarization rather than a proprietary review process.

The registry is a shared coordinate system, not a gatekeeper. `capdag.com` hosts a browsable registry of capability URNs and media types: a reference point for conventions, not a central ontology anyone must accept. Local registries and competing descriptions remain valid; the dispatch rule gives a formal basis for comparing and routing across them.

Platform-agnostic implementation. CapDAG is a protocol, not a platform. The specification is platform-agnostic by design, with Rust and Go implementations that can be adapted to other environments. A Linux or Windows port is a natural next step once the macOS product matures.

Local-first is not isolation. Cartridges currently run in heavily-sandboxed processes with no network access, but the protocol does not preclude a cartridge that proxies to a cloud service. Users can mix local and remote capabilities in the same pipeline with clear semantics.

7. Team

I'm Bahram, based out of Italy, with roughly a decade of production systems and data-infrastructure engineering: data warehousing and BI, large-scale content-analysis platforms on distributed storage and search infrastructure, and event-sourced microservice architectures. I have cofounded a startup before and led small technical teams with specialized roles (`www.linkedin.com/in/joharshamshiri`). The "we" in this document is aspirational: it reflects the team and community I intend to build, not the current headcount.

The surface area of MachineFabric (a custom frame protocol, a Rust and Swift dual-runtime execution model, XPC-isolated sandboxing, formally specified dispatch, and four shipping cartridges) is larger than a single engineer has historically been able to carry. Two factors make it tractable. First, modern AI development tooling (Claude Code, Codex, Antigravity, and others) produces the throughput of a small team rather than a single developer. Second, running small specialized startup teams before gives me a concrete model of the roles currently being occupied in parallel: systems engineer, protocol designer, data architect, native app developer, release manager. That model doubles as the hiring plan; each role becomes a real person as the project scales, against a description that already exists because it has been lived.

8. What We Are Not Claiming

To be explicit:

- We are not solving general semantic interoperability. CapDAG specifies one narrow thing, typed plugin dispatch, and does it rigorously.
- We are not replacing MCP, LangChain, or cloud agent frameworks. We occupy a different surface.

- We are not claiming the underlying mathematics is novel. It is standard type theory applied carefully; the value is rigor and disambiguation, not discovery.
- We are not claiming the open protocol alone is sufficient. The combination that matters is protocol, reference implementation, cartridge ecosystem, and a native user experience.

9. Conclusion

MachineFabric is a local-first macOS application for AI document processing. CapDAG is the open, typed capability-dispatch protocol it is built on. The protocol is small and specified; the architecture follows from it; the application is what someone actually opens; the timing tracks shifts in local-AI capability and privacy expectations that are now well underway. The narrower goal is what we are building toward: a practical, local-first foundation for typed composition, and the best application we can build on it.

A. CapDAG dispatch formalism

A capability URN names an input media type, an output media type, and a set of non-directional tags such as the operation, model family, language, or format. A media reference uses the same scheme. For example `cap:in="media:pdf";op=disbind;out="media:textable;page"`.

Formally, a Cap URN is a triple (i, o, y): input media type, output media type, and a bag of non-directional tags. For a provider $p = (i_p, o_p, y_p)$ and a request $r = (i_r, o_r, y_r)$:

$$\text{Dispatch}(p, r) \iff (i_r = \top \vee i_r \preceq i_p) \ \wedge \ (o_r = \top \vee o_p \preceq o_r) \ \wedge \ y_r \preceq y_p$$

- **Inputs are contravariant.** A provider may accept broader input types than the request supplies. A provider accepting `media:textable` can service a request sending `media:textable;page`, because `textable;page` is a refinement of `textable`.
- **Outputs are covariant.** A provider must produce output at least as specific as the request requires. A provider guaranteeing `media:json;textable;record` satisfies a request for `media:json`.
- **Tags are refinement-matched.** The provider must satisfy every explicit request constraint; it may refine the ones the request omits.

The relation is reflexive, transitive, and asymmetric: a more specific provider can service a more general request, but not the reverse. This is standard function subtyping applied to plugin matching.

Informal plugin systems tend to accumulate edge-case decisions (generic vs. specialized providers, partial matches, overlapping capabilities, fallback ordering, dispatch ambiguity), and each implementation derives them independently. CapDAG specifies them once. The mathematics is textbook; the value is that the rule is the same for every implementation, which makes routing inspectable and makes third-party cartridges portable.

B. Bifaci protocol and routing internals

CapDAG routes frames between the engine and a cartridge process through three layers:

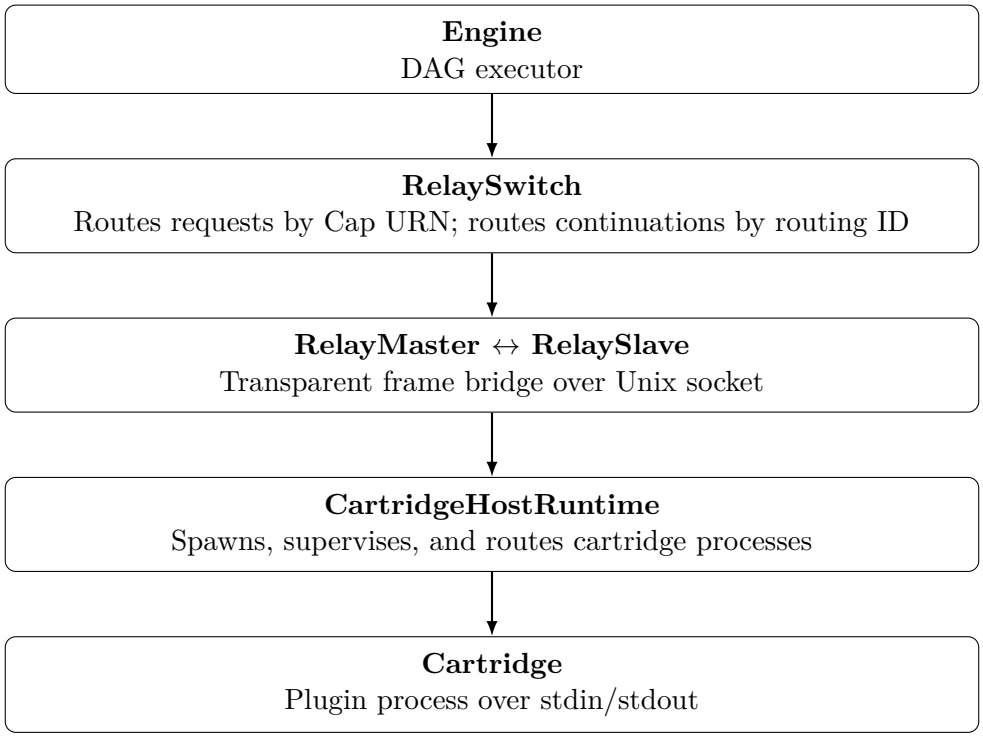

Figure 2: MachineFabric routing topology

The **RelaySwitch** is the central multiplexer: on each engine invocation it identifies eligible providers, picks the most specific one, assigns a routing ID, and records the mapping so response frames find their way back. **RelayMaster/RelaySlave** pairs form a transparent frame bridge over Unix sockets; most frames pass through unchanged, with two types intercepted to advertise capabilities upward and provide host resources (such as model paths) downward. The **CartridgeHostRuntime** spawns cartridges on demand, performs the HELLO handshake, routes requests and continuations by `(routing_id, request_id)` pairs, handles peer invocations between cartridges, and monitors health with 30-second heartbeats. Every connection is validated by a nonce echo (`CAP_IDENTITY` with a deterministic payload) before live traffic flows; a mismatch is fatal.

Cartridges invoke one another through peer invocation: a cartridge sends a REQ, the host records the request ID, forwards it to the switch, and routes the response back by matching the recorded ID. Composition is transitive without tight coupling. Protocol v2 supports multiplexed streaming responses with chunk indexing and checksums, plus relay-aware routing IDs.

0.6257.83979