

MachineFabric & CapDAG

Una piattaforma local-first per documenti e il protocollo di plugin tipizzato che la alimenta

machinefabric.com | github.com/machinefabric

Aprile 2026

Rivisto a ottobre 2026

Sommario

MachineFabric è un'applicazione local-first per il trattamento dei documenti su macOS, Linux e Windows. Gli utenti scelgono un'azione Transmute per uno o più file; su macOS l'azione è disponibile anche dal Finder. Il sistema identifica il tipo media dell'input, trova un percorso attraverso i plugin disponibili (le "cartucce") fino alla destinazione richiesta ed esegue quel percorso. Le cartucce locali elaborano sulla macchina dell'utente; una cartuccia che usa un servizio remoto resta una capacità di rete esplicita, non un passaggio implicito del planner.

Il substrato è **CapDAG**, un protocollo aperto per il dispatch tipizzato di capacità. Estende la composizione in stile Unix con un protocollo binario a frame per lo streaming di media tipizzati su pipe. Una capacità è nominata da una URN tagged che dichiara cosa accetta, cosa produce, un insieme di tag di raffinamento (operazione, lingua, modello, formato) e una coordinata d'effetto che descrive come si propaga l'identità dei media a runtime. Una richiesta è una domanda posta alle capacità registrate. Può lasciare aperto un lato, e la risposta è data dal subtyping standard delle funzioni (input controvarianti, output covarianti, effetti abbinati esattamente salvo wildcard) su ciò che la richiesta effettivamente chiede. Le regole sono scritte una volta sola, come modello verificato da una macchina, e ogni implementazione esegue codice generato da quel modello: le cartucce che si sovrappongono si risolvono allo stesso modo in ogni linguaggio.

Questo documento copre il problema, l'architettura, come una richiesta trova la sua capacità, il modello in cui le regole sono dimostrate e il protocollo wire. L'Appendice A enuncia le regole formalmente.

1. Il problema

Trattare documenti localmente con l'AI richiede tipicamente di assemblare strumenti eterogenei: estrazione da PDF, OCR per le pagine scansionate, modelli di embedding e modelli linguistici per riassunto, classificazione o Q&A.

Ogni componente espone un'interfaccia, un formato dati e un modello di esecuzione diversi. La composizione richiede codice di colla, di solito sotto forma di script o applicazioni ad hoc. Queste pipeline sono difficili da riutilizzare, ispezionare o estendere; una pipeline che funziona per i PDF di un utente raramente si trasferisce alle registrazioni audio di un altro senza riscrivere le giunture.

Le soluzioni esistenti affrontano parti del problema ma non offrono uno strato di composizione generale. I framework cloud assumono esecuzione di rete e orchestrazione centralizzata. Gli strumenti locali offrono funzionalità isolate ma mancano di un meccanismo per l'interoperabilità tipizzata. I sistemi di plugin integrati nelle singole applicazioni risolvono la composizione per quell'applicazione ma non si possono comporre fra applicazioni diverse.

Manca un sistema in cui le capacità locali possano essere composte in modo prevedibile, con tipi espliciti e senza lavoro di integrazione per ogni pipeline: una pipe Unix per trasformazioni di media tipizzate piuttosto che per stream di byte.

2. Panoramica del sistema

2.1. MachineFabric: il prodotto

MachineFabric ha un client nativo SwiftUI su macOS, un client Electron su Linux e Windows e un unico motore di esecuzione Rust condiviso. Gli utenti selezionano file e invocano un'azione "Transmute"; il sistema identifica il tipo media e instrada l'input attraverso una sequenza di capacità.

Per esempio, dato un video e la destinazione "trascrivi," MachineFabric trova il percorso (estrae l'audio, lo decodifica, lo manda allo speech-to-text) e mostra la rotta prima di partire. L'utente sceglie la destinazione, non i passi, e installare una cartuccia dà ai file esistenti nuove destinazioni. Ogni passo riporta l'avanzamento in tempo reale.

Le cartucce sono eseguibili autonomi che implementano trasformazioni specifiche. Ciascuna dichiara l'input che accetta, l'output che produce e i tag di raffinamento. Il sistema risolve un percorso di esecuzione valido a partire dalle cartucce disponibili e lo esegue senza codice di colla scritto dall'utente. I file restano sul disco dell'utente e l'inferenza gira sulla macchina dell'utente; senza connessione di rete, solo il download dei modelli non è disponibile.

Un esempio minimale:

```
document.pdf (media:ext=pdf)
→ pdfcartridge [cap:in="media:ext=pdf";disbind;
                out="media:enc=utf-8;ext=txt;page;plain-text"]
→ testo pagina (media:enc=utf-8;ext=txt;page;plain-text)
→ cartuccia modello [cap:in="media:enc=utf-8";summarize;
                    out="media:enc=utf-8;ext=txt;plain-text"]
→ riassunto (media:enc=utf-8;ext=txt;plain-text)
```

2.2. CapDAG: il protocollo

CapDAG è la specifica aperta di come le capacità sono nominate, tipizzate e abbinate alle richieste, e di come un host e una cartuccia si parlano. È indipendente da MachineFabric: qualsiasi runtime, in qualsiasi linguaggio, può essere conforme implementando i contratti URN, di planning e Bifaci applicabili. Rust è l'implementazione di riferimento del runtime; i mirror Go, Python, JavaScript e Swift/Objective-C ne implementano i sottoinsiemi documentati. JavaScript offre URN e planning ma non è un host per cartucce.

Tre idee reggono il protocollo.

- **Una capacità è una descrizione.** La sua URN nomina un tipo di media in input, un tipo di media in output e un insieme di tag non direzionali (operazione, modello, formato, lingua, e così via). Anche i tipi di media sono URN tagged, ordinate per raffinamento: `media:ext=pdf` è un `media:`, e una pagina di testo UTF-8 è un testo UTF-8.
- **Una richiesta è una domanda.** Dice cosa il chiamante ha e cosa vuole, e può tacere l'uno o l'altro. Le capacità che le rispondono si trovano con la regola standard per sostituire una

funzione a un'altra: un candidato può prendere più di quanto la richiesta invia e deve dare almeno quanto alla richiesta serve.

- **La regola esiste una volta sola.** È un modello nel dimostratore di teoremi Lean, in cui le proprietà su cui il sistema fa affidamento sono dimostrate come teoremi, e il codice che decide un abbinamento in ciascuno dei cinque linguaggi è generato da quel modello.

Il protocollo contribuisce una sola regola dimostrata, identica in ogni implementazione, così il routing è ispezionabile e riproducibile e le cartucce di terze parti hanno un contratto stabile. La Sezione 4 segue la regola da una singola richiesta a una rotta completa, la Sezione 5 descrive il modello e l'Appendice A la enuncia formalmente.

3. Architettura

MachineFabric separa il client desktop, il motore Rust condiviso, l'host delle cartucce e i processi cartuccia. Il client comunica con il motore via gRPC. Su macOS, **CartridgeDaemon** scopre e ospita le cartucce installate e le collega al motore; su Linux e Windows, **unifloom-daemon** svolge lo stesso ruolo. Il motore possiede routing, planning, esecuzione, persistenza SQLite e RelaySwitch Bifaci. Il client possiede presentazione, mediazione dell'accesso ai file, supervisione dei processi e integrazione con la piattaforma.

Le cartucce sono eseguibili autonomi in qualsiasi linguaggio. Comunicano con l'engine tramite **Bifaci**, un protocollo a frame su stdin/stdout con stream multiplexati, handshake, verifica dell'identità, controllo di flusso e keepalive. La topologia di routing e i dettagli del protocollo sono nell'Appendice B.

L'engine costruisce un grafo di capacità a partire dalle cartucce registrate all'avvio, e il planner trova un percorso al suo interno quando l'utente richiede un'operazione. Il grafo è limitato, gli stessi input si risolvono sempre nello stesso percorso, e il percorso è ispezionabile nell'app. Si tratta di pathfinding, non di pianificazione AI generica.

4. Da una richiesta a una rotta

Tutto ciò che l'engine fa con le capacità è una sola operazione: porre una domanda all'insieme delle capacità registrate. Instradare una chiamata, completare una capacità scritta a metà nell'editor, elencare ciò che un file può diventare e pianificare una rotta a più passi sono la stessa operazione, posta lasciando aperte parti diverse. Questa sezione la segue da una richiesta a una rotta completa.

4.1. Le capacità descrivono; le richieste chiedono

Una capacità è un impegno: prende questo e dà quello. Una richiesta non è una capacità. È una domanda, e una domanda può lasciare un lato non chiesto: “cosa mi dà una trascrizione di questo, qualunque cosa le serva?”, “in cosa si può trasformare questo file?”, “quale potrebbe essere il passo intermedio?”

Su un lato che la richiesta dichiara vale la direzione consueta. L'input del candidato deve coprire ciò che la richiesta invia: un riassuntore che prende qualsiasi testo UTF-8 serve una richiesta che ha in mano una pagina di testo UTF-8. L'output del candidato deve rientrare in ciò che la richiesta vuole: un candidato che promette un record JSON serve una richiesta di un record, e uno che

promette soltanto “qualche media” non serve nessuna richiesta che abbia bisogno di qualcosa in particolare.

Un lato che la richiesta non dichiara è *ignoto*, e ignoto non è il tipo “qualsiasi cosa”. Una capacità il cui input è `media:` ha fatto un’affermazione su ogni tipo: li prende tutti. Una richiesta che non ha detto cosa invierà non ha affermato nulla, quindi su quel lato ogni candidato passa. È questa distinzione a far sì che un solo meccanismo serva tutte le domande di sopra. Una destinazione da sola è una richiesta con l’input ignoto; una capacità astratta come `cap:convert-image` sta per ogni conversione concreta che ne porta i tag; una URN di capacità scritta in parte si completa nelle capacità compatibili con quanto è stato scritto fin lì.

4.2. Garantito, possibile e completo

Due descrizioni si possono confrontare in due modi: l’una va bene per l’altra *sempre*, oppure *potrebbe*? Una capacità che dà “qualche immagine” potrebbe dare un JPEG, e non è garantito che lo faccia. CapDAG tiene separate le due risposte. Una chiamata è instradata solo su una garanzia. Una ricerca riporta una possibilità come possibilità, e ogni risposta porta un grado: esatto, garantito, possibile o escluso. Il planner agisce sui primi due; l’interfaccia può mostrare il terzo come qualcosa che vale la pena provare.

Una terza distinzione è fra una descrizione e una cosa. Un *tipo* di media che non menziona un tag non ne dice nulla. Un *valore* che esiste, e l’elenco dei tag propri di una capacità, sono completi: non hanno i tag che omettono. Perciò i dati che arrivano a runtime sono confrontati con un tipo dichiarato per la cosa finita che sono, e una richiesta di una capacità *senza* un certo tag è servita da ogni capacità che non lo porta, senza che nessuno debba dichiararne l’assenza.

4.3. Scegliere fra i candidati

Quando più cartucce servono la stessa richiesta, vince quella che calza meglio. Si prendano due estrattori di pagine, uno dichiarato per input PDF come `cap:disbind;in="media:ext=pdf";out="media:enc=utf-8;ext=txt;page;plain-text"` e uno che accetta `media:.` Il primo è scelto quando l’input è un PDF, e quello generico resta disponibile per tutto il resto. La misura è la specificità, uno score numerico sui tag di una URN che non diminuisce mai per raffinamento. Fra i candidati che servono una richiesta è scelto quello la cui specificità è più vicina a quella della richiesta: prima uno score uguale, poi il candidato più specifico più vicino, poi quello più generale più vicino; a parità resta il primo registrato.

La validità viene prima della preferenza: un candidato che non serve la richiesta non è mai considerato, per quanto alto sia il suo score. La selezione è dimostrata corretta (ciò che restituisce serve la richiesta), completa (restituisce qualcosa ogni volta che qualcosa serve) e ottimale (nulla di ciò che serve è classificato strettamente meglio). È questo a rendere l’ecosistema di cartucce componibile come gli strumenti Unix: una cartuccia specializzata si installa senza rompere quella generica, e ciascuna è scelta per ciò che viene effettivamente chiesto.

4.4. Rotte nel grafo delle capacità

Quando l’utente chiede una destinazione che nessuna singola capacità serve (“riassumi questo PDF”), il planner cerca una rotta: PDF → `disbind` → testo della pagina → `summarize` → riassunto. Il grafo in cui cerca è fatto delle stesse domande. “Cosa prende questo?” dà gli archi uscenti da un

tipo, “cosa dà questo?” gli archi entranti, e una capacità fluisce nella successiva quando ciò che dà è garantito essere qualcosa che la successiva prende. Una rotta è una catena di tali passi, e le capacità che stanno su qualche rotta fra due tipi sono le scelte per un passo intermedio.

La ricerca è limitata dal grafo finito e fissato al momento della registrazione. La rotta viene mostrata all’utente prima dell’esecuzione, e questa è una proprietà del planner, non una funzionalità di UI: il planner produce un’espressione in Machine Notation che nomina ogni passo, ogni input e ogni tipo di output atteso, e la UI la rende visibile. Se una destinazione non è raggiungibile, per esempio per un modello mancante o un formato intermedio non supportato, il planner lo dice per nome. Non esiste uno stato opaco di “non si può fare”.

4.5. Raffinamento al momento del valore

Alcune decisioni di routing si possono prendere solo quando un valore esiste, perché un valore dice più del suo tipo dichiarato. Un cap dichiara il suo argomento di model-spec come `media:enc=utf-8;llm;model-spec`, ma la stringa effettiva (`hf:meta-llama/Llama-3.2-1B-Instruct`, `hf:mlx-community/...`, `hf:mistralai/Mistral-7B-Instruct`) determina quale adattatore di prompt formatting il runtime applica. CapDAG gestisce questo tramite un registry di adattatori di valore: una URN di base più una stringa vengono raffinate in una URN più specifica, e il dispatch sceglie l’adattatore della famiglia con la regola ordinaria.

5. Un modello, cinque implementazioni

La regola di abbinamento di CapDAG è scritta una volta sola, come modello nel dimostratore di teoremi Lean. Il modello definisce cosa significa ogni forma di tag, quando una descrizione ne raffina un’altra, quando due potrebbero descrivere la stessa cosa, cosa chiede una richiesta, quali capacità le rispondono, come si compongono le rotte e come si sceglie la risposta migliore.

Le proprietà su cui il sistema fa affidamento sono teoremi di quel modello, non aspettative: il raffinamento è un preordine; equivalenza significa “lo stesso insieme di tag”; una garanzia implica una possibilità; una richiesta con un lato ignoto è servita esattamente dalle capacità che servono qualche richiesta completamente tipizzata che essa potrebbe diventare; una domanda più stretta ha meno risposte; le rotte si compongono; la selezione è corretta, completa e ottimale. Il modello non ammette enunciati non dimostrati né assiomi aggiunti.

La parte eseguibile del modello è compilata in Rust, Go, Python, JavaScript (come WebAssembly) e Swift/Objective-C da `lungo`, un generatore di codice aperto per Lean, e versionata accanto a ciascun mirror. Parsing e stampa restano nativi di ogni linguaggio; le decisioni sono del modello. Una tabella di conformità scritta dal modello, con decine di migliaia di righe che coprono ogni coppia di forme di tag e ogni lettura di una richiesta, è rieseguita attraverso il parser di ciascun mirror nella sua stessa suite di test, cosicché un mirror che diverge fallisce indicando la riga.

6. L’ecosistema delle cartucce

Terze parti possono scrivere cartucce. Sono eseguibili autonomi in qualsiasi linguaggio; il formato manifest è JSON; il protocollo wire è aperto e piccolo; il layout di installazione è a livello di filesystem (`{root}/{slug}/v{registry_version}/{channel}/{name}/{version}/`) cosicché le installazioni sono ispezionabili e rimuovibili senza uno strumento speciale.

6.1. Sandboxing

Su macOS le cartucce girano in un servizio XPC separato sia dall'engine sia dall'app rivolta all'utente. MachineFabric non carica mai codice di cartuccia nel proprio processo. La sandbox XPC restringe le capacità delle cartucce di default: niente rete, niente accesso ai file dell'utente al di fuori di ciò che è esplicitamente consegnato. Una cartuccia che ha bisogno di accesso di rete (per esempio una cartuccia cloud-proxy) dichiara l'entitlement e viene revisionata prima di essere accettata nel manifest canonico.

Le cartucce distribuite devono essere code-signed con un certificato Developer ID, notarized da Apple e impacchettate come installer `.pkg` firmato. Per lo sviluppo l'host accetta cartucce non firmate piazzate direttamente nella root di installazione.

6.2. Il registry

CapDAG ha un registry pubblico e navigabile di URN di capacità e tipi di media: un punto di riferimento per le convenzioni, non un'ontologia centrale che chiunque debba accettare. I registry locali e descrizioni concorrenti restano valide; la regola di dispatch dà una base formale per confrontarle e instradare attraverso di esse. `cartridge-shelf` è la inbox pubblica di submission per le cartucce; `capfab` è l'equivalente per le definizioni di capacità e media def. Entrambe sono guidate da issue GitHub, revisionate da umani, e fuse nei manifest canonici che spediscono con ogni installazione.

6.3. Local-first non è isolamento

Le cartucce oggi girano senza accesso di rete di default, ma il protocollo non esclude una cartuccia che faccia da proxy verso un servizio cloud. Gli utenti possono mescolare capacità locali e remote nella stessa pipeline con semantica chiara: la regola di dispatch, il grafo del planner e l'avanzamento in tempo reale si comportano in modo identico. L'unica differenza è dove l'inferenza gira fisicamente.

7. Cosa non stiamo affermando

- Non stiamo risolvendo l'interoperabilità semantica generale. CapDAG specifica una cosa stretta, il dispatch tipizzato di plugin, e lo fa con rigore.
- Non stiamo rimpiazzando MCP, LangChain o i framework di agenti cloud. Occupiamo una superficie diversa: pipeline di trattamento file in cui l'unità di lavoro è una trasformazione di media tipizzata, non una chiamata di tool dentro una sessione di agente.
- Non stiamo affermando che la matematica sottostante sia originale. Ogni distinzione che il modello traccia è stata tracciata prima di noi, nella teoria dei tipi, nel matching di specifiche e nella rappresentazione della conoscenza. Il valore è il rigore e la disambiguazione, non la scoperta.
- Non stiamo affermando che il protocollo aperto da solo basti. La combinazione che conta è protocollo, implementazione di riferimento, ecosistema di cartucce ed esperienza utente nativa.

8. Conclusione

MachineFabric è un'applicazione desktop local-first per il trattamento dei documenti, costruita su CapDAG, un protocollo aperto per il dispatch tipizzato di capacità. La regola di abbinamento è specificata una volta sola, dimostrata in Lean e compilata in ogni implementazione.

A. Formalismo di dispatch CapDAG

Questa appendice enuncia le regole della Sezione 4 come le enuncia il modello. Ogni affermazione indicata come legge è un teorema del modello Lean.

A.1. Le URN tagged e il loro significato

Una URN tagged è uno schema e un insieme di tag, per esempio `media:enc=utf-8;ext=txt;page;plain-text`. Un tag vincola una chiave, e una chiave si trova in uno di due generi di stato: non impostata, oppure impostata a un valore. Ogni forma scritta significa l'insieme degli stati che ammette:

Forma	Ammette
$x=v$	impostata a v
x	impostata, a qualsiasi valore
$x!=v$	impostata, a un valore diverso da v
$x?=v$	non impostata, o impostata a un valore diverso da v
$!x$	non impostata
$?x$, o nessuna menzione di x	ogni stato

Una forma significa la stessa cosa da entrambi i lati di un confronto. Una URN denota quindi un insieme, le assegnazioni di stati alle chiavi che ciascuno dei suoi tag ammette, e le relazioni fra URN sono relazioni fra quegli insiemi.

A.2. Tre relazioni

- **Raffinamento**, $a \preceq b$: tutto ciò che a descrive, b lo descrive. È la garanzia. Leggi: è un preordine, si decide chiave per chiave, e il raffinamento reciproco vale esattamente quando i due insiemi di tag sono gli stessi.
- **Incontro**, $a \bowtie b$: qualcosa è descritto da entrambe. È la possibilità. Leggi: è riflessivo e simmetrico, il raffinamento lo implica, e *non* è transitivo, perché un PDF incontra “ha una qualche estensione”, che incontra un PNG.
- **Soddisfazione**, $t \models b$: una cosa completa t è descritta da b . I media di un valore e i tag propri di una capacità sono completi, quindi si leggono con ogni chiave omessa non impostata, e poi si confrontano per raffinamento.

La specificità è la somma degli score per tag (da 0 per una chiave non vincolata a 5 per una chiave richiesta non impostata). Legge: se $a \preceq b$ allora lo score di a è almeno quello di b .

A.3. Le richieste come query

Una capacità è una quadrupla (i, o, y, e) : tipo di media in input, tipo di media in output, tag non direzionali e una coordinata d'effetto sull'identità dei media a runtime, come in `cap:disbind;in="media:ext=pdf";out="media:enc=utf-8;ext=txt;page;plain-text"`.

Una query chiede qualcosa di ciascun lato di una capacità: che sia *entro* un tipo, che *copra* un tipo, che sia *lo stesso* tipo, oppure nulla (*ignoto*). Una URN di richiesta $r = (i_r, o_r, y_r, e_r)$ si legge come la query che chiede all'input del candidato di coprire i_r , o nulla quando i_r è lasciato aperto, e al suo output di essere entro o_r . Un candidato $p = (i_p, o_p, y_p, e_p)$ la serve quando

$$\text{Dispatch}(p, r) \iff (i_r \text{ aperto} \vee i_r \preceq i_p) \wedge o_p \preceq o_r \wedge (e_r = ? \vee e_p = e_r) \wedge y_p \models y_r.$$

- **Gli input sono controvarianti.** Un candidato può accettare input più larghi di quelli che la richiesta fornisce. Un candidato che accetta `media:enc=utf-8` serve una richiesta che invia `media:enc=utf-8;ext=txt;page;plain-text`.
- **Gli output sono covarianti.** Un candidato deve produrre un output almeno tanto specifico quanto la richiesta esige. Un candidato che garantisce `media:fmt=json;record` serve una richiesta di `media:record`.
- **Gli effetti corrispondono esattamente salvo wildcard esplicita.** Un effetto omesso significa `declared`; l'identità categorica è il cap esplicito `cap:effect=none`.
- **I tag sono soddisfatti.** Il candidato ha ogni tag che la richiesta chiede, può portare tag che la richiesta non menziona, e non ha i tag che omette.

La stessa URN letta come *pattern*, come quando si completa una capacità scritta in parte, chiede il duale: l'input del candidato entro quello del pattern, e il suo output che copre quello del pattern, o nulla quando l'output del pattern è aperto.

L'ignoto non è il tipo top. Legge: una query con input ignoto ammette esattamente i candidati che qualche query tipizzata ammette,

$$\text{admits}(\langle \text{ignoto}, o, y, e \rangle, p) \iff \exists i. \text{admits}(\langle \text{copre } i, o, y, e \rangle, p).$$

Un ignoto dice “per qualche tipo”; il tipo top dice “per ogni tipo”.

A.4. Gradi e selezione

Sostituendo in una query ogni raffinamento con un incontro si ottengono i candidati che la query *potrebbe* ammettere. Il grado di un candidato rispetto a una query è *esatto* quando è ammesso con ogni lato chiesto uguale a ciò che è stato chiesto, *garantito* quando è ammesso, *possibile* quando soltanto potrebbe esserlo, e *nessuno* altrimenti. Legge: ammesso implica potrebbe essere ammesso, quindi i gradi sono ordinati.

La selezione considera solo i candidati che servono la richiesta. Lo score di una capacità pesa il suo output più del suo input e questo più dei suoi tag; un candidato con score almeno pari a quello della richiesta è preferito a uno con score minore, una distanza minore a una maggiore, e a parità una registrazione precedente. Leggi: il candidato selezionato serve la richiesta; ne è selezionato uno

ogni volta che qualcuno serve; nessuno che serve è classificato strettamente meglio; e registrare un candidato che non serve non cambia nulla.

A.5. Cosa si compone

Il dispatch mette in relazione una capacità con una domanda, quindi non è la relazione che si compone. Tre lo sono:

- **Fra capacità come descrizioni.** a può stare al posto di b quando $i_b \preceq i_a$, $o_a \preceq o_b$, $y_a \preceq y_b$ e gli effetti concordano. È il subtyping delle funzioni, ed è un preordine.
- **Fra query.** Una query più stretta ammette meno candidati, transitivamente; e stabilire un lato ignoto restringe.
- **Fra capacità da un capo all'altro.** a fluisce in b quando $o_a \preceq i_b$. Una rotta da A a B è una catena in cui la prima capacità prende A , ciascuna fluisce nella successiva e l'ultima dà qualcosa entro B . Leggi: ogni tipo ha la rotta vuota verso sé stesso; le rotte si compongono; e una rotta da A a B è una rotta da qualsiasi cosa più stretta di A a qualsiasi cosa più larga di B .

L'ultima è la categoria in cui il grafo delle capacità è navigato: i tipi ne sono gli oggetti, le rotte le frecce, i raffinamenti le identità.

B. Protocollo Bifaci e dettagli di routing

CapDAG instrada i frame fra l'engine e un processo cartuccia attraverso tre strati:

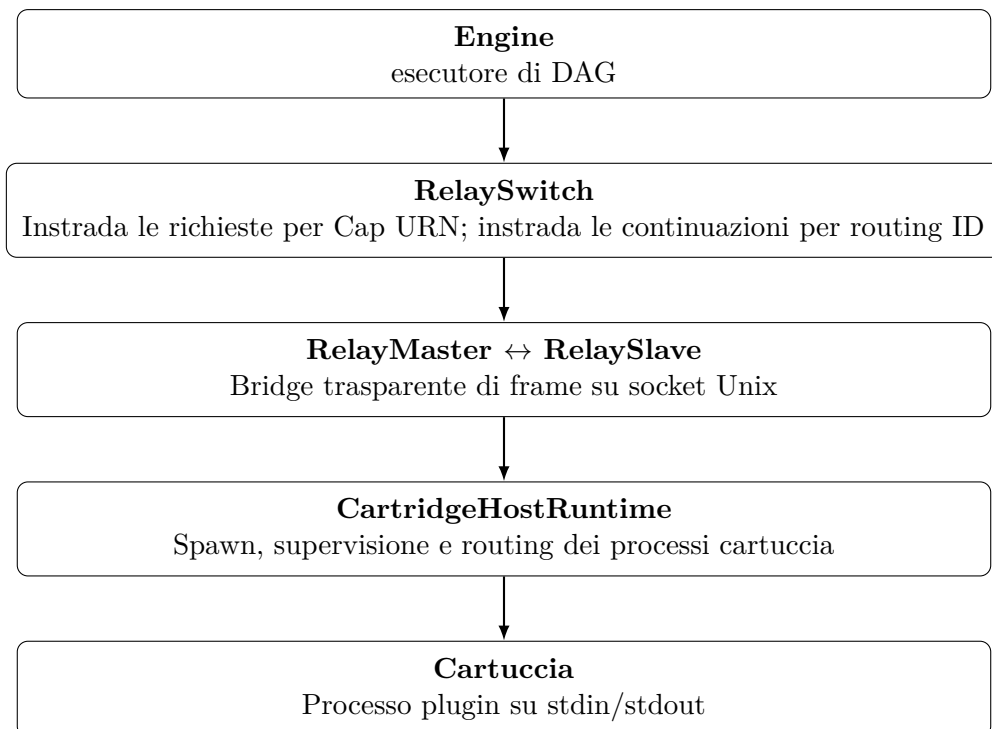


Figura 1: Topologia di routing di MachineFabric

Il **RelaySwitch** è il multiplexer centrale: a ogni invocazione dell'engine trova i candidati che servono la richiesta, sceglie fra loro come nell'Appendice A, assegna un routing ID e registra l'associazione cosicché i frame di risposta trovino la strada di ritorno. Le coppie **RelayMaster/RelaySlave** formano un bridge trasparente di frame su socket Unix; la maggior parte dei frame passa invariata, con due tipi intercettati per pubblicare le capacità verso l'alto e fornire risorse dell'host (come i percorsi dei modelli) verso il basso. Il **CartridgeHostRuntime** avvia le cartucce su richiesta, esegue l'handshake HELLO, instrada richieste e continuazioni per coppie (`routing_id`, `request_id`), gestisce le invocazioni peer fra cartucce e monitora la salute con un heartbeat ogni 30 secondi a cui va risposto entro 10. Ogni connessione è validata da un nonce echo (`CAP_IDENTITY` con un payload deterministico) prima che fluisca traffico vero; un mismatch è fatale.

Le cartucce si invocano l'un l'altra tramite peer invocation: una cartuccia invia un REQ, l'host registra il request ID, lo inoltra allo switch e instrada la risposta indietro abbinando l'ID registrato. La composizione è transitiva senza accoppiamento stretto.

Il protocollo è alla versione 4, che entrambe le parti dichiarano nell'handshake. Trasporta risposte streaming multiplexate con indicizzazione dei chunk e checksum, routing ID consapevoli del relay e un controllo di flusso a crediti per stream: un mittente può trasmettere solo tanti chunk quanti il ricevente ne ha concessi, cosicché un consumatore lento rallenta il suo produttore invece di esaurire la memoria.