Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

Un Framework Event-Sourced per AutoML Tracciabile basato su Riscrittura di Bigrafi: Progettazione, Implementazione e Proprietà Formali

Relatore:
Prof.
FABIO TOSI

Correlatore:
Prof.
STEFANO MATTOCCIA

Candidato:
ARSAL-HANIF LIVOROI

III Sessione
Anno Accademico 2024–2025

Abstract

Automated Machine Learning (AutoML) riguarda l'automazione, totale o parziale, delle attività di progettazione, configurazione e valutazione di modelli e processi di apprendimento automatico. In questa tesi, esso viene considerato soprattutto come esplorazione di configurazioni candidate — architetture, iperparametri e scelte di processo — tramite modifiche discrete e successive valutazioni. Nella pratica, tale ciclo sperimentale è spesso difficile da riprodurre e da verificare: le varianti architetturali si accumulano in configurazioni divergenti, la provenienza di metriche e artefatti non è sempre esplicita, e risulta complesso attribuire un miglioramento o un degrado a una specifica decisione progettuale.

La tesi presenta un framework per AutoML *tracciabile*, basato su *event sourcing* e su una rappresentazione intermedia dell'architettura esprimibile tramite *bigrafi* e relative trasformazioni. L'unità sperimentale primaria è il *trial*, che raccoglie struttura del modello, iperparametri di training, stato parametrico e riferimenti agli artefatti prodotti durante l'esecuzione.

L'evoluzione di ciascun trial viene registrata come log append-only di eventi, dal quale stato e viste derivate possono essere ricostruiti tramite *replay deterministico*. Le modifiche risultano così osservabili e confrontabili rispetto a viste mirate, mentre l'integrazione con strumenti esterni avviene tramite un'interfaccia bidirezionale di export/import con metadati di provenienza, così da preservare tracciabilità e riproducibilità. Infine, la strategia di ricerca resta intercambiabile: un'esplorazione guidata manualmente può essere sostituita da una procedura automatica senza modificare la semantica del dominio.

Keywords Automated Machine Learning (AutoML), tracciabilità sperimentale, event sourcing, bigrafi, riscrittura di bigrafi, riproducibilità.

Indice

Capitolo 1

Introduzione

1.1 Contesto

Automated Machine Learning (AutoML) può essere interpretato come un processo di esplorazione di configurazioni candidate — architetture, iperparametri e scelte di processo — tramite trasformazioni discrete e valutazioni. Una panoramica ampia dei metodi e dei sistemi AutoML è presentata nel volume curato da Hutter et al. [11] e nella survey di He et al. [10]. In ambito deep learning, la ricerca dell'architettura di un modello, o *neural architecture search* (NAS), è spesso organizzata attorno a uno spazio di ricerca, una strategia di ricerca e un meccanismo di stima delle prestazioni [5].

Nella pratica quotidiana della ricerca (ad es. in computer vision), questo ciclo sperimentale è iterativo: si definisce una baseline, si allena per un budget, si modifica una sola ipotesi alla volta e si confrontano alternative. Quando le modifiche sono strutturali (architetturali) e coinvolgono strumenti/ambienti diversi, diventa cruciale poter attribuire in modo affidabile *che cosa* è cambiato e *quando*.

Questa prospettiva, guidata dal ricercatore, è anche quella applicativa privilegiata nella presente tesi. Pur collocandosi nella cornice AutoML, il focus attuale del framework è soprattutto su ricerca architetturale e ottimizzazione degli iperparametri in contesti di ricerca sperimentale in computer vision, dove si parte da una baseline, si derivano varianti controllate, si osservano metriche e artefatti, e si confrontano traiettorie sperimentali. In particolare, il contesto applicativo che ha guidato molte scelte narrative del lavoro è vicino ai filoni del CVLab dell'Università di Bologna su stereo matching, depth estimation e adattamento dei modelli; una discussione più puntuale dei punti di contatto è rimandata alla sezione 3.1.

1.2 Problema

Senza un'infrastruttura deterministica e verificabile, l'esplorazione sperimentale diventa difficile da riprodurre, confrontare e spiegare, specialmente quando coinvolge trasformazioni strutturali e strumenti esterni. I problemi tipici includono:

- **Riproducibilità.** Ricostruire una traiettoria sperimentale richiede spesso ricostruire script, configurazioni e dipendenze; piccole differenze di ambiente rendono fragile la replica.
- **Comparabilità.** Confrontare alternative richiede una baseline comune e una descrizione precisa di "che cosa è cambiato"; l'uso di flag o rami di codice tende a produrre configurazioni divergenti e poco verificabili.
- **Integrazione di strumenti esterni.** Strumenti esterni possono enumerare o applicare trasformazioni e produrre artefatti; senza un contratto esplicito di export/import e metadati di provenienza è difficile mantenere coerenza e tracciabilità.

1.3 Contributo e idea chiave

Questa tesi propone un framework per AutoML tracciabile, basato su *event sourcing*, ossia sulla registrazione append-only degli eventi che descrivono l'evoluzione sperimentale. L'unità sperimentale primaria è il *trial*, mentre la componente architetturale del modello è rappresentata tramite una rappresentazione intermedia (IR) basata su bigrafo, dove un bigrafo è una struttura che descrive insieme relazioni di annidamento e di connessione tra componenti. In termini intuitivi, un trial raccoglie l'architettura corrente, gli iperparametri, i riferimenti ai pesi o ai checkpoint, lo stato di processo, le osservazioni derivate e il *lineage*, ossia il grafo di derivazione che collega tra loro i vari trial. Il modello in senso stretto può essere quindi letto come una vista derivata del trial, centrata sulla coppia architettura–pesi. La storia del trial è registrata come log append-only di eventi; il *replay deterministico*, che ricostruisce lo stato a partire dal log, produce lo stato corrente e le proiezioni derivate.

Le modifiche strutturali sono espresse come trasformazioni sull'IR basata su bigrafo, appoggiandosi alla letteratura sui *Bigraphical Reactive Systems* (BRS), ovvero sistemi di riscrittura basati su bigrafi, e alle pratiche di modellazione e tooling contemporanee [1], [13], [14]. Per supportare analisi mirate, il framework rende disponibili confronti rispetto a diverse viste (ad es. architettura, iperparametri, modello allenato), così da rendere esplicito sotto quale prospettiva due trial vengono considerati uguali o diversi.

Un aspetto centrale della progettazione è che la strategia di ricerca sia *intercambiabile per costruzione*: un ricercatore o una procedura automatica emettono lo stesso linguaggio di comandi; il dominio resta deterministico e la provenienza viene registrata nei metadati degli eventi.

Rimando alla vista d'insieme. Il flusso complessivo (controller, core deterministico, log degli eventi, backend, interfaccia di scambio con l'esterno e artefatti) è riassunto in Figura 6.1, introdotta nel Capitolo 6.

1.4 Obiettivi e non-obiettivi

Obiettivi. Gli obiettivi della tesi sono:

- definire una rappresentazione esplicita e trasformabile dell'architettura del modello (IR a bigrafo) e una dinamica di modifica basata su riscrittura;
- rendere la ricerca tracciabile tramite event sourcing, replay deterministico e provenance (incluso roundtrip verso strumenti esterni);
- supportare confronti mirati tra trial tramite confronti rispetto a una vista e derivazione di varianti a baseline comune;
- progettare un'architettura che separi core deterministico ed effetti, consentendo l'uso di backend esterni e l'estendibilità verso policy automatiche.

Non-obiettivi. Non sono obiettivi di questa tesi:

- proporre una nuova strategia di stato dell'arte per l'AutoML, per l'ottimizzazione degli iperparametri (*Hyperparameter Optimization*, HPO) o per la ricerca dell'architettura (NAS): la policy è esterna e intercambiabile;
- massimizzare le prestazioni su benchmark come obiettivo primario: la valutazione è orientata a proprietà verificabili dell'infrastruttura;
- includere parallelismo o asincronia come requisito corrente, considerate estensioni future.

1.5 Struttura della tesi

Il Capitolo 2 introduce il dominio e il modello mentale della ricerca (trial, viste sul modello, attori e flusso di lavoro). Il Capitolo 3 raccoglie i concetti di base: elementi essenziali di AutoML, bigrafi e *Bigraphical Reactive Systems* (BRS), event sourcing e una nozione di *boundary*, intesa come interfaccia verso l'esterno, ispirata alle *lenses*, ossia a costrutti per descrivere accessi e aggiornamenti coerenti su strutture dati; colloca inoltre il framework rispetto a un contesto applicativo privilegiato di ricerca in stereo e depth estimation. Il Capitolo 4 formalizza requisiti e criteri di successo. Il Capitolo 5 definisce la semantica a eventi e le proprietà verificabili (replay, correttezza del confronto rispetto a una vista, coerenza dell'interfaccia verso l'esterno). Il Capitolo 6 descrive l'architettura del framework (core deterministico, backend, *boundary* e modelli di lettura). Il Capitolo 7 presenta il flusso di lavoro e i casi d'uso guida (mutazioni interne, roundtrip, derivazione di trial). Infine, i capitoli successivi discutono implementazione di riferimento (Capitolo 8), valutazione property-driven (Capitolo 9) e prospettive future (Capitolo 11).

Capitolo 2

Dominio e modello mentale della ricerca AutoML

Questo capitolo introduce il *modello mentale* con cui interpretare il framework nel contesto della ricerca AutoML (*Automated Machine Learning*): quali sono gli oggetti del dominio, quale unità sperimentale viene tracciata, e quali attori interagiscono con il sistema. L'obiettivo non è ancora formalizzare in dettaglio la semantica del replay o i contratti architetturali, ma chiarire *che cosa* il framework considera rilevante dal punto di vista del ricercatore. La notazione formale è fissata in Appendice A e viene richiamata qui solo nella misura necessaria a stabilizzare il lessico. Le formule introdotte in questo capitolo hanno quindi valore di *modello concettuale*: servono a isolare le componenti rilevanti per la tesi, non a riprodurre in modo letterale i tipi concreti dell'implementazione. In altre parole, questo capitolo fissa il lessico e l'intuizione operativa con cui leggere il framework. La semantica formale completa verrà sviluppata solo più avanti.

2.1 Scenario base

Lo scenario è presentato dal punto di vista del ricercatore e riflette attività tipiche della ricerca ML: partire da uno stato iniziale esplicito, allenare per un budget definito, modificare una sola cosa alla volta, e confrontare alternative derivate da una baseline comune. L'obiettivo è rendere questo ciclo *riproducibile* e *attribuibile*: ricostruire una traiettoria sperimentale e spiegare che cosa è cambiato tra due varianti rilevanti. [8], [16]

Workflow del ricercatore. Nel caso più semplice il ricercatore: (i) definisce o seleziona una baseline, (ii) esegue training e valutazione per un budget noto, (iii) osserva metriche e artefatti, (iv) modifica intenzionalmente una sola componente, (v) confronta la nuova variante sperimentale con quella di partenza. Questo schema è vicino alla pratica del "one change at a time": la variazione strutturale o configurativa è esplicita, mentre il confronto avviene a posteriori sulle viste e sugli artefatti prodotti.

Trial come unità sperimentale dell'esperimento. Il framework non tratta il "modello" come un oggetto isolato, ma come parte di un *trial* inserito in un *esperimen-*

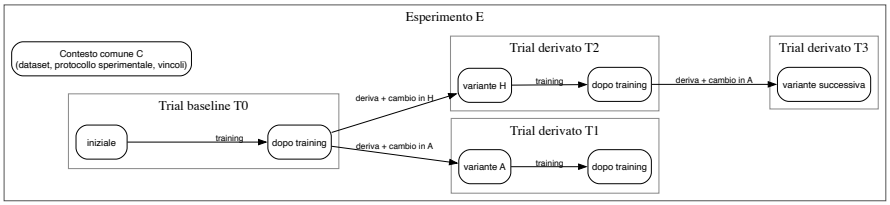


Figura 2.1: Struttura concettuale di un esperimento come contenitore di trial. A partire da una baseline comune, il framework consente di derivare varianti diverse mantenendo il contesto condiviso e il relativo grafo di derivazione (lineage) necessario a confrontarle.

to. Nel seguito useremo quindi due livelli complementari. Il primo è l'esperimento, inteso come contenitore complessivo del contesto di confronto e dell'insieme dei trial rilevanti:

$$E := \langle C, \mathcal{T} \rangle$$

dove C raccoglie il contesto comune dell'esperimento e $\mathcal{T}$ l'insieme dei trial che appartengono a quel contesto. Il secondo è il trial, che costituisce l'unità sperimentale concreta su cui il ricercatore opera:

$$T := \langle A, H, W, P, O, L \rangle$$

dove A è l'architettura, H raccoglie gli iperparametri, W rappresenta lo stato parametrico o il suo riferimento persistito, P descrive lo stato di processo locale al trial, O raccoglie le osservazioni prodotte durante la sua storia, e L ne cattura identità e lineage, ossia le informazioni minime del suo grafo di derivazione. Useremo inoltre la proiezione leggera

$$\text{core}(T) := \langle A, H, W \rangle$$

per riferirci al nucleo sperimentale del trial senza appesantire il testo. Quando invece servirà parlare del modello in senso stretto, useremo la vista derivata

$$M(T) := \langle A, W \rangle$$

che isola architettura e stato parametrico. Queste scritture hanno funzione concettuale: mettono in evidenza le componenti che contano per la ricerca sperimentale, senza pretendere di coincidere 1:1 con i tipi concreti mantenuti dal framework. In questa tesi il trial resta quindi l'unità sperimentale fondamentale: è l'oggetto che si crea, si allena, si deriva e si confronta, mentre l'esperimento fornisce il contesto comune entro cui tali trial acquistano significato comparativo. La Figura 2.1 sintetizza questa relazione tra esperimento, trial e derivazioni, mostrando come varianti nate dalla stessa baseline sono collegate attraverso un grafo di derivazione.

Bootstrap del trial: punti di partenza espliciti. Un esperimento inizia da uno stato iniziale dichiarato. Nel framework questo stato può essere ottenuto in tre modi concettuali:

- **Trial vuoto.** Il ricercatore parte da uno stato iniziale minimale del trial e costruisce progressivamente A e H tramite comandi.
- **Baseline predefinita.** Il ricercatore parte da una configurazione iniziale ripetibile (template o script), utile per confronti immediati.
- **Import e transfer learning.** Adattatori di import permetteranno di inizializzare A (e, quando possibile, W) da modelli esterni.

Derivazioni controllate a partire da una baseline comune. Per confrontare alternative a partire da una baseline comune, il ricercatore crea trial *derivati*: ottiene trial distinti che condividono uno stato base verificabile e divergono per una modifica intenzionale. Nel caso ideale, ogni ramo introduce una singola variazione rilevante — ad esempio una trasformazione di A oppure un aggiornamento di H — così da mantenere chiaro il legame tra cambiamento introdotto e comportamento osservato. Questo rende naturale il confronto tra ablation, incrementi controllati e strategie alternative di training. Informalmente si può ancora parlare di ramificazione tra trial, ma il punto concettuale centrale è la *derivazione di trial* all'interno dello stesso esperimento, con lineage esplicito, cioè con un grafo di derivazione verificabile, e baseline comune.

Che cosa significa "replicare" un trial. Nel contesto di questa tesi, replicare un trial significa ricostruire uno stato sperimentale confrontabile. In particolare, ciò implica recuperare il suo nucleo core(T), il suo stato di processo locale P, le osservazioni O utili per analisi e confronto, e le informazioni di lineage L, vale a dire di grafo di derivazione, necessarie a collocarlo nella storia dell'esperimento. Dal punto di vista del framework, ciò che il ricercatore osserva come trial corrisponde quindi a una lettura concettuale dello stato canonico ricostruito via replay della sua storia, non a una copia indipendente mantenuta separatamente. Il dettaglio di come questa ricostruzione avvenga tramite log e replay è rimandato a Capitolo 5; qui basta osservare che il framework è progettato per non separare il "modello" dalla sua storia sperimentale.

Osservazione e confronto. Dopo ogni blocco di training o dopo una modifica significativa, il ricercatore osserva metriche, curve e artefatti associati al trial. Il confronto tra alternative non riguarda quindi solo una struttura, ma una configurazione sperimentale completa con un certo nucleo core(T), un certo stato di processo locale, un certo grafo di derivazione e certi risultati osservati. In questa prospettiva, il framework è pensato per supportare non solo la costruzione del modello, ma soprattutto il ciclo *definisci, allena, deriva, confronta* come pratica di ricerca.

2.2 Il dominio: ricerca come evoluzione di trial tracciabili

Nel framework proposto la ricerca è modellata come evoluzione di trial tracciabili. Una traiettoria sperimentale non è descritta soltanto dal suo ultimo stato. Conta

anche la successione di decisioni e trasformazioni che ha portato a quello stato. Questa scelta consente di trattare in modo uniforme:

- la creazione di una baseline;
- la derivazione di varianti controllate;
- le modifiche su architettura, iperparametri e processo;
- l'associazione tra cambiamenti introdotti e risultati osservati.

Dal punto di vista del dominio, la ricerca viene quindi vista come una storia di *cambiamenti ammissibili* applicati a trial, più che come una semplice successione di esecuzioni numeriche. Questa impostazione favorisce tre proprietà che saranno formalizzate nei capitoli successivi:

- **Riproducibilità,** perché la traiettoria può essere ricostruita in modo deterministico a partire dai record persistiti;
- **Attribuibilità,** perché ogni variazione significativa può essere associata a una modifica esplicita e a una provenienza;
- **Confrontabilità,** perché trial derivati da una baseline comune possono essere analizzati come alternative della stessa storia sperimentale.

In questo senso, il dominio non decide *che cosa convenga provare*, ma definisce *quali cambiamenti siano leciti e tracciabili*. La strategia di ricerca — manuale o automatica — resta esterna al dominio: il framework offre un linguaggio operativo comune per esprimere e registrare tali scelte.

Cambiamenti applicati e confronti osservati: intuizione operativa Nel seguito useremo la distinzione tra un cambiamento *applicato* e un confronto *osservato*. Intuitivamente, il primo corrisponde a un passo della storia del trial, mentre il secondo corrisponde al confronto tra stati sotto una certa vista. Questa distinzione è centrale per la tesi, ma la sua formalizzazione dettagliata è rimandata a Capitolo 5. Nel presente capitolo è sufficiente fissare l'idea di fondo: il framework registra cambiamenti controllati e rende poi possibili confronti mirati tra varianti.

2.3 Scomposizione concettuale del trial

Per orientare la discussione successiva conviene esplicitare quali componenti del trial siano rilevanti per la tesi. La scomposizione che segue non introduce ancora la semantica completa dello stato canonico, ma chiarisce quali aspetti dello stato sperimentale devono restare osservabili, confrontabili e tracciabili.

Architettura (A) come rappresentazione intermedia basata su bigrafo. In questa tesi A è trattata principalmente come rappresentazione intermedia (IR) dell'architettura del modello, basata su bigrafo. Un bigrafo combina contenimento (place graph) e connessioni (link graph), risultando adatto a rappresentare sia modularità (blocchi, sottoreti, componenti composti) sia connettività (skip connection, fusioni, diramazioni e collegamenti non puramente gerarchici). [13] La semantica di flusso non è assunta implicitamente: ruoli, porte e attributi rendono esplicite precondizioni e vincoli di validità. Quando è necessario ispezionare o manipolare bigrafi con tool esterni, l'implementazione di riferimento usa BigraphER come ambiente di riferimento per riscrittura e simulazione. [21]

Iperparametri (H). Gli iperparametri sono modellati come configurazione versionabile e confrontabile. Nel prototipo, H può partire come mappa chiave→valore per semplicità, con possibile evoluzione verso rappresentazioni più tipizzate per optimizer, scheduler e altre scelte di training. Il punto essenziale, a livello di dominio, è che gli iperparametri siano trattati come parte esplicita del trial, non come dettaglio implicito esterno al suo stato sperimentale.

Pesi (W) e checkpoint. I pesi non vengono memorizzati direttamente nel log del dominio: W rappresenta lo stato parametrico del trial, mentre la sua materializzazione concreta avviene tramite checkpoint o artefatti prodotti dal backend. Questa scelta permette di distinguere in modo pulito tra stato canonico della storia e dati numerici pesanti, mantenendo comunque la possibilità di confrontare: (i) stessa architettura con stati parametrici diversi, (ii) architetture diverse, (iii) architettura uguale ma iperparametri o protocollo differenti. Quando servirà isolare il modello in senso stretto, useremo la vista derivata $M(T) := \langle A, W \rangle$, lasciando fuori da tale vista gli iperparametri H.

Processo, osservazioni e lineage (P, O, L). P descrive lo stato di processo locale al trial: budget, epoche, fasi, seed e altri elementi necessari a interpretarne l'esecuzione. O raccoglie le osservazioni prodotte nella sua storia: metriche, checkpoint, marker di fase, provenienza e altri riferimenti utili al confronto. L rende invece esplicito il lineage minimo del trial, vale a dire il suo grafo di derivazione minimo: identità, eventuale trial genitore e sorgente immediata della derivazione. Questa distinzione è importante perché separa: (i) il *come procede* localmente un trial, (ii) *che cosa viene osservato* durante la sua storia, (iii) *da dove proviene* il trial nel contesto dell'esperimento. Il protocollo sperimentale condiviso e gli altri elementi di contesto comune restano invece al livello dell'esperimento, cioè dentro C.

Osservazione. Il formalismo a bigrafo è generale e potrebbe modellare anche dati, risorse o strumenti in modo più esteso. In questo elaborato, però, il focus resta su architettura e trasformazioni strutturali, con iperparametri, processo, osservazioni e lineage come contesto essenziale per la ricerca sperimentale. Questa delimitazione consente di mantenere il modello sufficientemente espressivo senza disperdere il contributo principale della tesi.

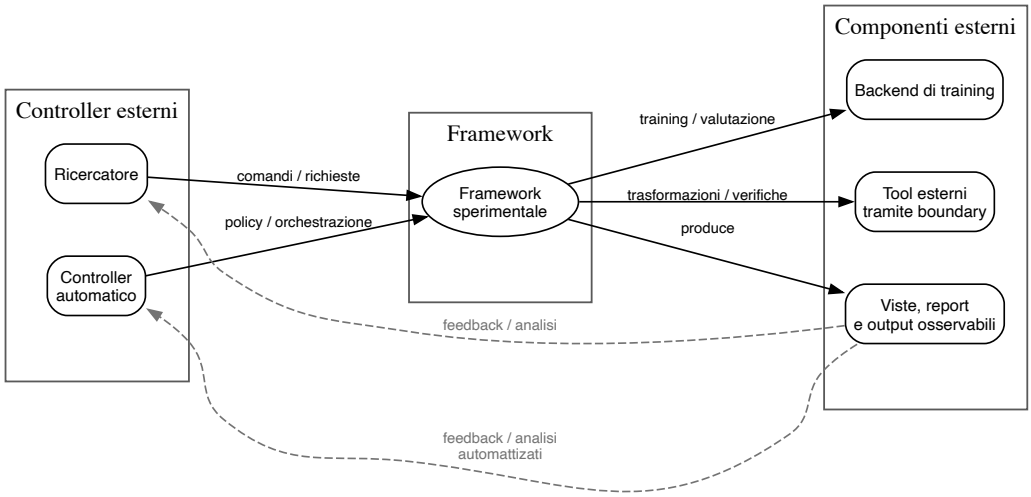


Figura 2.2: Quadro concettuale degli attori e delle componenti esterne rilevanti per il framework: il ricercatore e un eventuale controller automatico interagiscono con il framework, che orchestra il lavoro, delega training e valutazione a backend esterni, coinvolge tool esterni tramite il boundary, ossia l'interfaccia verificabile verso l'esterno, quando servono trasformazioni o verifiche, e produce viste e report utili al confronto sperimentale.

2.4 Attori e responsabilità

Controller della ricerca. Decide quale comando inviare al sistema: creare un trial, derivarne uno nuovo, applicare una modifica, avviare training o richiedere un'interazione con il *boundary*, inteso qui come interfaccia verificabile verso l'esterno. Il controller può essere il ricercatore stesso oppure una policy esterna che orchestra la ricerca.

Domain (core deterministico). Valida i comandi, produce eventi accettati, evolve lo stato tramite replay e impone invarianti di buona-formazione.

Backend. Interpreta l'IR A in un modello eseguibile e gestisce training e valutazione, producendo artefatti quali checkpoint e metriche.

Tool esterni. Operano su viste esportate del modello o di sue porzioni e rientrano nel framework tramite il boundary verificabile appena introdotto.

La Figura 2.2 riassume questi ruoli e rende esplicito il confine tra controller della ricerca, core di dominio, backend e tool esterni.

Questa separazione riflette una scelta metodologica precisa: il controller *sceglie* cosa tentare, il dominio *giudica e registra* ciò che è ammesso, il backend *esegue* la parte numerica, e gli strumenti esterni collaborano solo attraverso interfacce controllate. Ne risulta un quadro in cui la ricerca può restare flessibile sul piano operativo, senza perdere coerenza semantica e tracciabilità.

Rimando ai capitoli successivi. La semantica a eventi e le proprietà verificabili sono descritte in Capitolo 5; l'architettura a layer e i confini tra core ed effetti sono discussi in Capitolo 6; i workflow e i casi d'uso concreti (UC1–UC3) sono presentati in Capitolo 7.

Capitolo 3

Background tecnico

Questo capitolo introduce i concetti necessari a collocare la tesi rispetto alla letteratura. L'obiettivo non è fornire una trattazione esaustiva, ma fissare un vocabolario comune e motivare le scelte che verranno specializzate nei capitoli successivi. In particolare, richiameremo quattro filoni rilevanti: *Automated Machine Learning* (AutoML); i bigrafi, formalismi grafici che combinano contenimento e connettività, insieme ai *Bigraphical Reactive Systems* (BRS); l'event sourcing, in cui lo stato viene ricostruito da una storia append-only di eventi, in stile *Domain Modeling Made Functional* (DMMF); e il problema del *view update* affrontato dalla letteratura sulle *lenses*, o lenti bidirezionali.

3.1 Contesto applicativo di riferimento del framework

Pur non essendo specializzato su uno specifico task di computer vision, il framework proposto è stato progettato assumendo come riferimento applicativo privilegiato flussi di lavoro sperimentali vicini a quelli della ricerca in stereo matching e depth estimation del CVLab dell'Università di Bologna. Più precisamente, tale contesto non rappresenta un vincolo di dominio del framework, bensì il principale scenario che ne ha guidato gli obiettivi, i criteri di progettazione e alcune priorità architetturali. Una sintesi recente del settore mostra infatti che la stereo vision contemporanea non ruota soltanto attorno alla proposta di nuove architetture, ma anche attorno a sfide quali robustezza fuori dominio, generalizzazione zero-shot, adattamento online e valutazione su benchmark particolarmente difficili. [26]

Adattamento e mutazioni controllate. Una parte rilevante della produzione del gruppo ruota attorno all'adattamento del modello durante il deployment o in regime di adattamento continuo, come mostrano lavori su *Real-Time Self-Adaptive Deep Stereo* e *Continual Adaptation for Deep Stereo*. [17], [24] In scenari di questo tipo, il problema non consiste soltanto nell'allenare una rete, ma anche nel mantenere interpretabile la traiettoria di cambiamenti che conduce da una baseline a una variante adattata. Il framework proposto nella tesi si colloca bene in questo spazio perché rende espliciti trial, derivazioni, *lineage* — inteso qui come grafo di derivazione

—, mutazioni architetturali e metadati di provenienza, offrendo così una struttura naturale per studiare strategie di adattamento senza perdere la storia sperimentale.

Scenari difficili, benchmark e comparabilità. Un altro asse rilevante del lavoro del CVLab riguarda la robustezza della depth estimation in condizioni difficili, ad esempio in presenza di superfici speculari o trasparenti, come nel benchmark Booster, e più di recente in approcci orientati alla robustezza zero-shot come Stereo Anywhere. [2], [19] In questi casi diventa particolarmente importante poter confrontare in modo rigoroso varianti ottenute da una baseline comune, isolare una modifica alla volta e conservare evidenza riproducibile di metriche, checkpoint, artefatti e trasformazioni applicate. Questa esigenza si accorda con il modello mentale assunto nella tesi: il framework non promette di risolvere di per sé il problema della depth estimation robusta, ma si propone di fornire un'infrastruttura con cui organizzare in modo più controllato la ricerca sperimentale che lo affronta.

Provenienza di modelli, dati e protocolli. Filoni come *NeRF-Supervised Deep Stereo* mostrano inoltre che, in computer vision, spesso non evolve soltanto il modello, ma anche il modo in cui viene costruito il segnale di supervisione e, più in generale, il protocollo con cui si ottengono i dati di training. [27] Da questo punto di vista, un framework basato su event sourcing risulta interessante perché può registrare in modo uniforme non solo training e mutazioni del trial, ma anche la provenienza delle decisioni, il ruolo di strumenti esterni e la relazione tra configurazione del modello e protocollo sperimentale adottato. Il contributo della tesi va quindi letto soprattutto come un *livello meta-sperimentale*: non un nuovo metodo di stereo o depth estimation, ma una struttura per descrivere, riprodurre e confrontare in modo esplicito l'evoluzione di modelli e procedure.

Collegamento con il lavoro svolto da CVLab. Il collegamento con il CVLab va quindi inteso in due sensi complementari. Da un lato, le esigenze concrete emerse in quel contesto — gestione di baseline e varianti, mutazioni controllate, grafo di derivazione (*lineage*) esplicito, integrazione con strumenti esterni, confronto riproducibile di metriche e artefatti — hanno costituito la principale guida applicativa per definire il framework e valutarne le scelte progettuali. Dall'altro, il prototipo attuale non va letto come una piattaforma già in grado di sostenere integralmente l'intera pipeline di ricerca del gruppo: alcuni aspetti restano ancora parziali, prototipali oppure demandati a componenti esterni. In questo senso, il contributo della tesi consiste soprattutto nell'aver fissato le fondamenta semantiche e architetturali — trial, event sourcing, mutazioni tracciabili, una nozione di *boundary* come interfaccia verso l'esterno, viste ed export — sulle quali un supporto più completo a pipeline di questo tipo potrà essere costruito ed esteso in futuro.

3.2 Automated Machine Learning (AutoML)

In senso ampio, AutoML può essere visto come un processo di esplorazione di configurazioni candidate sotto vincoli di costo, come tempo, risorse di calcolo e

memoria. Una configurazione comprende tipicamente scelte di modello, iperparametri, aspetti di processo (budget, criteri di stop, checkpointing) e dettagli legati alla valutazione. Una sintesi autorevole di metodi e sistemi AutoML è riportata nel volume *Automated Machine Learning: Methods, Systems, Challenges* [11]; una survey utile per inquadrare i sottoproblemi lungo la pipeline è [10].

Pipeline e obiettivi di automazione. In modo compatibile con la letteratura, una pipeline AutoML include almeno: (i) preparazione dei dati, (ii) feature engineering, (iii) generazione del modello (scelta della famiglia/architettura e degli iperparametri), (iv) validazione e selezione, e (v) gestione dell'esperimento (logging, versionamento, riproducibilità). [10] In questa tesi usiamo tale tassonomia come riferimento pratico, ma restringiamo il contributo alla parte di *generazione/trasformazione del modello* e, soprattutto, alla *gestione tracciabile dell'esperimento*.

Definizione operativa. Nel seguito considereremo un sistema *AutoML* se include almeno:

- uno **spazio di ricerca**: cosa può cambiare (architettura, iperparametri, setup di training);
- un **controllore di ricerca**: come vengono proposte e selezionate le mosse (manuale o automatica);
- una **procedura di valutazione**: metriche e protocollo per confrontare candidati;
- **gestione delle risorse**: budget, vincoli o criteri equivalenti (es. early stopping);
- **gestione dell'esperimento**: tracciamento sufficiente per riprodurre e verificare le decisioni.

Questa scomposizione è utile perché separa chiaramente: (i) *che cosa* può variare, (ii) *chi o cosa* decide le mosse, (iii) *come* le alternative vengono valutate, e (iv) *quale evidenza* resta disponibile per riproduzione e verifica.

Sottocampi rilevanti. Nel filone classico, il problema *Combined Algorithm Selection and Hyperparameter optimization* (CASH), che combina selezione dell'algoritmo e ottimizzazione degli iperparametri, è rappresentato da lavori come Auto-WEKA [23]. In ambito deep learning, la ricerca dell'architettura (*neural architecture search*, NAS) viene spesso descritta tramite *search space*, *search strategy* e *performance estimation* [5]; sistemi AutoML pratici integrano inoltre componenti di valutazione e gestione degli esperimenti (ad es. auto-sklearn) [6]. Per la tesi, questo quadro è importante soprattutto per distinguere il livello della *strategia di ricerca* dal livello della *rappresentazione tracciabile delle configurazioni e delle loro trasformazioni*.

Ricerca dell'architettura (NAS) e relazione con la tesi

La letteratura su NAS organizza il problema secondo tre componenti: *search space*, *search strategy* e *performance estimation*. [5] Questa scomposizione è utile per chiarire il perimetro del progetto:

- **Search space.** La tesi si concentra soprattutto su come rappresentare in modo esplicito lo spazio delle trasformazioni architetturali e delle variazioni sperimentali rilevanti.
- **Search strategy.** La strategia che decide *quale* mossa eseguire non è il contributo principale: può essere manuale oppure automatica, purché resti osservabile e confrontabile.
- **Performance estimation.** La valutazione resta necessaria per confrontare candidati, ma il focus della tesi non è proporre una nuova procedura di stima; è piuttosto rendere le valutazioni tracciabili e interpretabili nel contesto dell'esperimento.

Trial e traiettorie sperimentali. In molti sistemi AutoML l'unità sperimentale è un *trial*: una traiettoria di trasformazioni e valutazioni che porta da una configurazione iniziale a una o più configurazioni derivate. Questa prospettiva è utile perché sposta l'attenzione dal singolo file di configurazione alla storia sperimentale che rende interpretabili confronti, varianti e baseline comuni.

Perimetro del progetto rispetto alla pipeline AutoML. Il contributo di questa tesi si concentra principalmente su: (i) generazione del modello nel senso di rappresentazione e trasformazioni architetturali, (ii) supporto a variazioni di iperparametri e processo, e soprattutto (iii) *gestione dell'esperimento* tracciabile. Sono invece fuori perimetro l'automazione end-to-end della preparazione dei dati e del feature engineering, così come lo sviluppo di una strategia di ricerca di stato dell'arte, che restano estensioni possibili su una infrastruttura di sperimentazione più controllabile.

3.3 Bigrafi e Bigraphical Reactive Systems (BRS)

I bigrafi sono formalismi grafici che combinano due strutture:

- **place graph**: rappresenta relazioni di contenimento/annidamento (composizione, modularità);
- **link graph**: rappresenta relazioni di connessione (dipendenze e collegamenti tra componenti).

Questa duplicità è utile per modellare architetture ML moderne, che hanno sia una struttura modulare (blocchi, sottoreti, componenti riusabili) sia una connettività non necessariamente ad albero (skip connections, multi-branch, fusioni).

Struttura duale (place vs link). La distinzione tra contenimento e connessioni costituisce il riferimento intuitivo che useremo nei capitoli successivi quando parleremo di rappresentazione intermedia (IR) strutturale e di regole di reazione.

Scopo dei bigrafi secondo Milner. Milner introduce i bigrafi come un formalismo per modellare sistemi di agenti in cui contano sia la *località* ("chi è dentro cosa", contesto/spazio) sia la *connettività* ("chi è collegato a chi", interazioni). [13], [14] La dinamica è descritta da regole di reazione (BRS) che trasformano il bigrafo in passi discreti. [13]

Strumenti e analisi. Per l'uso pratico, BigraphER fornisce strumenti open-source per definire signature, bigrafi e regole, ed eseguire riscritture e simulazioni in modo riproducibile. [21] Inoltre, esistono strumenti per esplorare automaticamente lo spazio delle transizioni e verificare proprietà (ad es. model checking) per modelli bigrafici. [15] Una guida moderna orientata alla modellazione pratica (ad esempio località delle regole, condizioni, priorità e varianti operative) è discussa in [1].

Modellare architetture ML con bigrafi (intuizione)

Nel contesto di questa tesi, un'architettura può essere letta come una struttura in cui: (i) i *controls*, ossia i tipi di componenti o moduli, (ii) il *place graph*, che cattura composizione e modularità, (iii) il *link graph*, che cattura connettività e dipendenze. Questa rappresentazione rende naturale esprimere mutazioni come riscritture: una *reaction rule* specifica un pattern da riconoscere (*redex*) e la struttura risultante (*reactum*). [13]

Osservazione. Per la tesi, il punto essenziale non è il dettaglio sintattico dello strumento, ma il fatto che un formalismo bigrafico permette di rappresentare insieme modularità e connettività, e di esprimere trasformazioni locali in modo dichiarativo. I dettagli con cui tali trasformazioni vengono poi integrati nel framework saranno discussi nei capitoli successivi.

3.4 Event sourcing e Domain Modeling Made Functional

L'event sourcing è un pattern architetturale in cui la fonte di verità non è lo stato corrente, ma una sequenza append-only di eventi che descrive come lo stato sia evoluto nel tempo. Lo stato corrente è ottenuto applicando gli eventi in ordine (replay), rendendo naturale la riproducibilità e la verifica del percorso che ha portato allo stato corrente. [8]

In stile Domain Modeling Made Functional (DMMF), la logica di dominio è modellata come trasformazioni pure:

- un `Command` rappresenta un'intenzione (azione richiesta);

- `decide` valida il comando rispetto allo stato corrente e produce una lista di `Event` (o un errore);
- `evolve` applica un evento allo stato;
- il replay è un fold (`fold evolve`) sull'intera sequenza di eventi.

Questa separazione permette di confinare gli effetti (training, I/O, strumenti esterni) fuori dal core deterministico. [28]

Una vista complessiva del flusso (comandi, eventi, replay) e del confine degli effetti è riportata in Figura 6.1 (Capitolo 6).

Osservazione. Per la tesi, questo background è rilevante perché offre una grammatica pulita per distinguere intenzioni, eventi, replay ed effetti esterni. La specializzazione di questa grammatica al dominio dei trial e delle trasformazioni architetturali verrà definita nei capitoli successivi.

3.5 Lenti bidirezionali, complement e *boundary*

L'integrazione con strumenti esterni introduce un problema tipico: esportare una vista di uno stato interno, trasformarla esternamente e reintegrarne il risultato preservando coerenza e riproducibilità. La letteratura sulle *lenses* affronta il *view-update problem*: come sincronizzare in modo consistente una vista e una sorgente, spesso richiedendo un *complement* che conserva l'informazione residua necessaria a rendere deterministico il ritorno (putback). [7]

In questa tesi, tale prospettiva è utile come sfondo concettuale per ragionare su export/import e roundtrip verificabili verso strumenti esterni. Il punto importante, a questo livello, non è una teoria completa delle lenses, ma l'idea che una vista esportata da sola possa non bastare a ricostruire in modo univoco lo stato sorgente senza informazione aggiuntiva. I dettagli del *boundary* adottato dal framework, inteso come interfaccia bidirezionale verso l'esterno, verranno poi precisati nei capitoli dedicati.

Capitolo 4

Requisiti e criteri di successo

Questo capitolo raccoglie i requisiti del framework proposto e i criteri con cui valutarne correttezza e utilità. I requisiti sono formulati con taglio ingegneristico, vale a dire in termini di ciò che il framework deve supportare e rendere possibile, mentre la semantica che li rende verificabili viene definita nel Capitolo 5.

L'obiettivo non è ottimizzare un singolo algoritmo di *Automated Machine Learning* (AutoML), ma rendere *tracciabile* e *riproducibile* l'evoluzione di un trial: è un'esigenza ricorrente nello sviluppo di sistemi ML, dove complessità del workflow e "debito tecnico nascosto" tendono ad accumularsi se i cambiamenti non sono espliciti e confrontabili.[20] Inoltre, la riproducibilità sperimentale è un tema riconosciuto nella comunità ML e motivazione per pratiche di reporting e gestione degli artefatti.[16]

4.1 Scopo, assunzioni e perimetro

Scopo. Il framework supporta ricerca AutoML *tracciabile*: un ricercatore in modalità manuale o un controller automatico esplorano varianti di architettura e iperparametri attraverso modifiche discrete, osservabili e ricostruibili. Il contributo primario non è una nuova strategia di ricerca, ma un'infrastruttura che rende: (i) esplicita la dinamica di modifica della configurazione sperimentale, inclusa la componente architetturale rappresentata tramite bigrafi, formalismi che combinano contenimento e connettività di un grafo;[13] (ii) riproducibile l'evoluzione tramite *event sourcing*, dove lo stato viene ricostruito a partire da una storia append-only di eventi;[8] (iii) verificabile l'interazione con backend e tool esterni tramite tracciabilità, provenienza e confronti osservazionali.[18]

Assunzioni principali.

- Il log eventi è append-only e ordinato; lo stato di un trial è derivato via replay.
- Il core di dominio (`decide`/`evolve`) è deterministico e privo di I/O, mentre gli effetti esterni sono separati.
- Training, validazione e interazione con tool esterni producono artefatti esterni referenziati dal sistema, non incorporati nel log come payload pesanti.

- L'architettura è rappresentata da una rappresentazione intermedia (IR) basata su bigrafo a livello di blocchi; eventuali estensioni oltre la sola architettura sono compatibili con l'impostazione del framework, ma non necessarie per lo scope corrente.

Fuori perimetro (non-obiettivi).

- Competizione con lo stato dell'arte di AutoML, dell'ottimizzazione degli iperparametri (*Hyperparameter Optimization*, HPO) o della ricerca dell'architettura (*neural architecture search*, NAS): le strategie di ricerca sono intercambiabili e trattate come estensione.
- Esecuzione distribuita o scheduling concorrente come requisito corrente (possibile sviluppo futuro).
- Dimostrazioni "proof-heavy": le proprietà sono intese come proprietà semantiche definite con precisione e verificabili, non come risultati di metateoria formale completa.

4.2 Requisiti funzionali (RF)

I requisiti funzionali sono organizzati per responsabilità: stato canonico, trasformazioni, tracciabilità, interfaccia verso l'esterno (*boundary*), backend, derivazione di trial e osservabilità. Quando un requisito usa un termine tecnico (es. *provenienza*, *confronto rispetto a una vista*, *roundtrip*), esso anticipa una nozione che verrà precisata nel capitolo successivo, senza ancora fissarne qui la formalizzazione completa.

Stato canonico e inizializzazione

RF1 — **Inizializzazione di un trial.** Il framework consente di inizializzare un trial a partire da una configurazione di base definita da specifica interna oppure tramite meccanismi di import, quando disponibili. L'oggetto iniziale include almeno la componente architetturale, gli iperparametri, lo stato dei pesi o i riferimenti iniziali agli artefatti pertinenti, il processo sperimentale e le informazioni minime di osservazione e *lineage*, inteso qui come grafo di derivazione, necessarie a collocare il trial nella storia complessiva.

RF2 — **Stato canonico ricostruibile.** In ogni momento lo stato di un trial è ricostruibile a partire dal log eventi; snapshot e proiezioni sono opzionali e rigenerabili.

Trasformazioni e dinamica

RF3 — **Trasformazioni interne su architettura.** Il framework consente di applicare trasformazioni architetturali interne espresse come riscritture sulla IR scelta per la componente architetturale.[13] Ogni trasformazione è validata rispetto a vincoli di correttezza strutturale e produce uno stato architetturale successivo confrontabile con il precedente.

RF4 — Modifiche di iperparametri e processo. Il framework consente di modificare iperparametri e aspetti di processo in modo tracciabile, supportando sequenze tipiche della ricerca: allenare per N epoche, modificare, allenare ancora, e confrontare gli esiti.

Tracciabilità, eventi e provenienza

RF5 — Eventi con provenienza esplicita. Il framework conserva, insieme agli eventi di dominio, metadati sufficienti a documentare provenienza, causalità e correlazione delle operazioni rilevanti. Tali informazioni devono restare distinguibili dal contenuto semantico del dominio, così da preservare stabilità del canone e tracciabilità dell'esecuzione.

RF6 — Provenienza dei comandi. Il framework conserva provenienza sufficiente per distinguere comandi emessi da un ricercatore rispetto a comandi emessi da un agente o da una policy automatica, senza alterare la semantica del dominio. Come riferimento concettuale per il "chi/cosa/quando/da quali input", si adotta il lessico di PROV-DM.[18]

Interfaccia verso l'esterno (*boundary*) e tool esterni

RF7 — Roundtrip verso tool esterni. Il framework supporta roundtrip verso tool esterni, ossia cicli di export e rientro: deve essere possibile esportare una vista esterna, reintegrare l'esito dell'elaborazione esterna e mantenere tracciabilità sufficiente a verificare la coerenza del rientro. Tale requisito è motivato dal view-update problem e dalla prospettiva delle *lenses*, costrutti per descrivere accessi e aggiornamenti coerenti su strutture dati.[7]

Backend e gestione degli effetti

RF8 — Backend esterni per training e valutazione. Il training e la valutazione sono delegati a backend esterni, in modo da mantenere il core di dominio indipendente dal meccanismo esecutivo concreto e compatibile con più ecosistemi ML.

RF9 — Tracciabilità degli effetti. Le operazioni non pure, come training, validazione ed export/import, devono essere tracciate in modo esplicito insieme ai loro esiti, producendo riferimenti ad artefatti esterni pertinenti (per esempio checkpoint, report o file di interscambio).

Derivazione di trial e confrontabilità

RF10 — Derivazione di trial con lineage. Il framework consente di derivare nuovi trial a partire da uno stato base comune, in modo da confrontare alternative in trial distinti mantenendo tracciabilità della derivazione e un *lineage* verificabile, ossia un grafo di derivazione ricostruibile.

Osservabilità e strumenti per la ricerca

RF11 — Confronti multi-dominio rispetto a una vista. Il framework rende disponibili confronti per architettura, iperparametri, processo e interfaccia verso l'esterno, calcolabili rispetto a viste o proiezioni pertinenti (per esempio la sola architettura oppure la configurazione completa), utilizzabili per ablazione, confronto e debugging.

RF12 — Metriche e visualizzazioni. Il framework consente di raccogliere metriche nel tempo e visualizzarle, collegandole a eventi significativi della timeline; inoltre consente di esportare visualizzazioni dell'architettura e, quando necessario, una vista a bigrafo. La motivazione pratica è coerente con quanto discusso nella letteratura sui sistemi per il ciclo di vita ML, dove tracciamento di parametri, artefatti e riproducibilità sono esigenze centrali.[29]

4.3 Requisiti non funzionali (RNF)

RNF1 — Determinismo e replay. Il replay del log eventi ricostruisce lo stesso stato, e le stesse proiezioni derivabili, a parità di log e di riferimenti agli artefatti pertinenti.

RNF2 — Verificabilità end-to-end della provenienza e della derivazione. Deve essere possibile rispondere a domande come: *chi ha emesso questo comando?*, *quale trasformazione ha prodotto questo stato?* e *da quale trial deriva questo risultato?*, collegando eventi, provenienza, artefatti e grafo di derivazione.

RNF3 — Separazione tra core puro ed effetti. La logica di dominio resta pura; I/O e computazioni esterne sono confinati in backend e tool.

RNF4 — Estendibilità. Devono essere aggiungibili nuovi backend, nuove regole e nuovi tool esterni senza cambiare la semantica del core e senza invalidare la leggibilità dei log preesistenti.

RNF5 — Usabilità per ricerca. Il framework deve supportare workflow tipici della ricerca: modifiche una alla volta, derivazione controllata di varianti, confronto tra traiettorie ed ispezione dei cambiamenti tramite confronti rispetto a una vista e visualizzazioni.

RNF6 — Overhead ragionevole. Logging, confronti rispetto a una vista e tracciabilità devono introdurre overhead contenuto rispetto al training; nella valutazione l'overhead viene discusso in modo qualitativo, evidenziandone i fattori dominanti e le possibili ottimizzazioni.

4.4 Criteri di successo (CS)

I criteri di successo trasformano i requisiti in proprietà verificabili e utili per casi d'uso di ricerca.

CS1 — Replay correctness. Replay completo produce lo stesso stato finale e proiezioni rigenerabili coerenti, incluse le informazioni necessarie a ritrovare gli artefatti pertinenti.

CS2 — Correttezza del confronto rispetto a una vista. I confronti sono coerenti con la traiettoria: rigenerarli a partire da due stati della storia deve restituire un'evidenza osservabile consistente con il cambiamento intervenuto, utile per ablation e debugging.

CS3 — Coerenza dell'interfaccia verso l'esterno. Il roundtrip esterno è verificabile: esportazione, rientro e controlli di coerenza producono un'evidenza tracciabile e ripetibile.

CS4 — Controller intercambiabile. Sostituire una gestione manuale con un controller automatico o con uno stub non richiede modifiche al dominio; cambia solo la provenienza dei comandi.

CS5 — Evidenza tramite casi d'uso. I casi d'uso guida (UC1–UC3) dimostrano in modo riproducibile trasformazioni interne, interfaccia verso l'esterno, derivazione di trial e osservabilità delle metriche.

4.5 Tracciabilità requisiti → evidenze

Per mantenere l'allineamento tra specifica, implementazione e valutazione, i requisiti sono collegati a evidenze concrete (report, test di replay, plot). La tabella seguente è una matrice minima di tracciabilità (requisiti → evidenze).

Area	**Requisiti**	**Evidenza (placeholder)**
Core basato su eventi	RF2, RF5, RNF1, CS1	Replay + confronto stato; rigenerazione proiezioni
Confronti	RF11, CS2	Report di confronto sui passi; rigenerazione sotto vista
Interfaccia esterna	RF7, CS3	Roundtrip esterno + controlli di coerenza
Derivazione	RF10, CS5	Grafo di derivazione + confronto tra trial derivati
Backend	RF8, RF9	Esecuzione training/valutazione + riferimenti ad artefatti
Osservabilità	RF12, RNF5	Grafici metriche legati a eventi; export architettura

Capitolo 5

Semantica e proprietà verificabili

Questo capitolo esplicita la semantica del framework con un obiettivo pragmatico: trasformare in oggetti *definiti* e *verificabili* nozioni che, nel lavoro sperimentale, vengono spesso trattate in modo informale. In particolare, fissiamo: (i) quale stato viene considerato canonico; (ii) come quello stato evolve tramite comandi, eventi e replay; (iii) come distinguere i *passi canonici* dai *confronti rispetto a una vista*; (iv) quali proprietà possono essere controllate in modo ripetibile.

La distinzione guida del capitolo è la seguente:

- un **evento canonico di dominio** è un fatto registrato nella storia del sistema e interpretato tramite replay;
- un **passo canonico** δ è una scorciatoia *derivata* che raggruppa uno o più eventi omogenei, senza sostituire il lessico primario `command/event/decide/evolve`;
- un **confronto rispetto a una vista** Δ_V è invece un confronto osservazionale tra trial rispetto a una vista V scelta dal ricercatore.

In altri termini, questo capitolo separa *ciò che accade nel dominio* da *ciò che risulta visibile sotto una vista*. Tale separazione evita di confondere la storia effettiva dell'esperimento e dei suoi trial con i confronti costruiti a fini di analisi. Per una classificazione operativa dei *passi canonici* e per un quadro sinottico dei *confronti rispetto a una vista* per dominio rimandiamo all'Appendice A.

5.1 Oggetti canonici e stato osservabile

Esperimento. L'unità semantica più generale del framework è l'*esperimento*:

$$E := \langle C, \mathcal{T} \rangle$$

dove C raccoglie il contesto comune di confronto e $\mathcal{T}$ l'insieme dei trial appartenenti a quel contesto. Questa notazione è utile perché consente di descrivere in modo uniforme sia i comandi locali a un trial sia quelli che introducono nuovi trial, come la creazione radice o la derivazione.

Trial. L'entità sperimentale concreta è però il *trial completo*, inteso come unità sperimentale primaria:

$$T := \langle A, H, W, P, O, L \rangle$$

dove: A descrive la componente architetturale del trial; H raccoglie la configurazione dichiarata o gli iperparametri; W rappresenta lo stato parametrico o il riferimento operativo ai pesi; P rappresenta lo stato di processo/runtime; O raccoglie osservazioni e materializzazioni del trial; L raccoglie identità e genealogia minima. Nel lessico adottato in questa tesi, il trial non è quindi un semplice contenitore del modello, ma l'unità che rende confrontabili addestramento, derivazione, osservazioni e lineage.

Core e modello in senso stretto. Quando conviene alleggerire la notazione, useremo il *core* del trial:

$$\text{core}(T) := \langle A, H, W \rangle$$

come proiezione espositiva delle componenti più direttamente coinvolte nelle trasformazioni del candidato. Useremo inoltre il *modello in senso stretto* solo come vista derivata del trial:

$$M(T) := \langle A, W \rangle.$$

Questa scelta evita di ricadere in una formulazione model-centric del tipo $M = \langle A, H, W \rangle$: nel quadro corrente, il soggetto semantico primario resta il trial completo, mentre il modello è una sua proiezione utile quando il focus è su architettura e stato parametrico.

Osservazioni e lineage. Le componenti O e L non sono dettagli accessori. In forma minima, il *lineage*, inteso qui come grafo di derivazione minimo del trial, è:

$$L := \langle id, parent, src \rangle$$

dove id identifica il trial corrente, $parent$ il trial padre se esiste, e src la sorgente immediata della derivazione quando disponibile. In modo analogo, O raccoglie metriche, checkpoint e provenance osservazionale. Ciò che nel testo può apparire come semplice reportistica entra quindi nella definizione stessa del trial, pur restando distinto dal suo core leggero.

Fonte di verità e viste. Lo stato corrente non è assunto come sorgente primaria della verità. La fonte canonica è la storia degli eventi del dominio; lo stato corrente di E o di un suo trial è il risultato del replay di quella storia. Le viste usate da CLI, notebook, report e confronti sono quindi *read models* derivati dallo stato ricostruito, non uno stato alternativo concorrente.

Due livelli da non confondere. A questo punto conviene rendere esplicita una distinzione che guiderà tutto il capitolo:

- il **livello canonico** descrive eventi, replay e stato ricostruito dell'esperimento e dei suoi trial;

- il **livello osservazionale** descrive proiezioni, confronti e report derivati da quello stato.

Il replay appartiene al primo livello; i *confronti rispetto a una vista*, i report e le viste appartengono al secondo.

Schema minimo di lettura. Per evitare sovrapposizioni con i capitoli successivi, conviene fissare fin d'ora una griglia minima di lettura:

- il **replay** ricostruisce lo stato del trial e dell'esperimento a partire dalla storia degli eventi;
- il **passo canonico** riassume in forma derivata una trasformazione di stato prodotta da `decide` e realizzata da `evolve`;
- il **confronto rispetto a una vista** legge differenze osservabili tra stati gia ricostruiti, senza mutarli;
- l'**interfaccia verso l'esterno** (*boundary*) descrive il contratto verificabile con cui alcune operazioni attraversano strumenti esterni e rientrano nella storia del trial.

Questa separazione è sufficiente, in questa sede, per mantenere distinto il livello semantico del framework dal workflow operativo, dal mapping implementativo e dalla valutazione per oracoli.

5.2 Semantica event-sourced: eventi, replay e determinismo del core

Il framework adotta una semantica basata su *event sourcing*: la verità del dominio è una storia append-only di eventi [8]. Il core segue lo stile *Domain Modeling Made Functional* [28]:

- un `Command` rappresenta un'intenzione;
- `decide` valida il comando rispetto allo stato corrente e produce una lista di `Event` oppure un errore;
- `evolve` applica un evento allo stato;
- il replay ricostruisce lo stato applicando `evolve` agli eventi nell'ordine registrato.

Semantica minimale. La forma più generale è experiment-scoped:

$$\texttt{decide} : (E,\ \texttt{Command}) \to \texttt{Result}([\texttt{Event}]) \qquad \texttt{evolve} : (E,\ \texttt{Event}) \to E$$

e il replay è il fold sugli eventi:

$$\texttt{replay}(E_0, [e_1, \ldots, e_n]) = \texttt{fold}(\texttt{evolve}, E_0, [e_1, \ldots, e_n]).$$

Questa formulazione è importante perché copre correttamente anche i casi in cui nasce un nuovo trial: `create` e `derive` non sono anomalie della semantica, ma casi normali del medesimo schema `decide/evolve`.

Vista locale sul trial. Quando il comando è naturalmente locale a un trial già esistente, la stessa semantica può essere letta anche come vista trial-scoped. In quella lettura, il replay ricostruisce il singolo trial a partire dal suo stato iniziale e dalla propria storia locale. Questa seconda lettura è utile soprattutto per `train`, `proc` e `obs`, ma non sostituisce la forma generale experiment-scoped.

Significato denotazionale. Sia $\bar{e} = [e_1, \ldots, e_n]$ una storia valida dell'esperimento. Il suo significato denotazionale è il replay deterministico degli eventi:

$$\text{ReplaySem}(\bar{e}) := \text{fold } \texttt{evolve}\ E_0\ \bar{e} = E_n.$$

Equivalentemente, ciascun evento induce una trasformazione di stato $\text{step}_e : E \to E$ e la storia corrisponde alla loro composizione ordinata. La semantica canonica dipende quindi dagli eventi e dal loro ordine, non dalla particolare UX che ha originato i comandi.

Famiglie di cambiamento. Nel lessico adottato nella tesi, le famiglie canoniche più utili per leggere la storia sono: `create`, `derive`, `train`, `proc` e `obs`. Esse non introducono una seconda semantica concorrente, ma classificano a posteriori gruppi omogenei di eventi di dominio. Il *passo canonico* verrà infatti introdotto proprio come nozione derivata da questa grammatica event-first.

Metadati, provenance ed effetti esterni. Gli eventi possono portare metadati di provenance e correlazione utili per audit, analisi o riproducibilità, in linea con l'idea generale di PROV-DM [18]; tali metadati non ridefiniscono però la semantica del core, che continua a dipendere dagli eventi canonici e dal loro ordine. In modo analogo, il replay non coincide con l'esecuzione numerica del training: il backend interpreta la rappresentazione intermedia (IR) ed esegue training, valutazione ed export come effetti esterni ricondotti a osservazioni, artefatti e boundary. In questo senso, il core resta deterministico rispetto agli eventi del dominio, mentre la variabilità infrastrutturale viene confinata nel livello osservazionale e nei boundary.

5.3 Passo canonico e confronto rispetto a una vista

La distinzione tra *passo canonico* e *confronto rispetto a una vista* è il punto concettuale più importante del framework. Un passo canonico descrive la storia del dominio; un confronto rispetto a una vista descrive ciò che risulta visibile quando quella storia viene proiettata sotto una certa osservazione.

Il passo canonico come nozione derivata

Dato un comando q, chiamiamo *passo canonico* la trasformazione di stato indotta dalla sequenza di eventi prodotta da `decide`. In forma experiment-scoped:

$$\delta_q(E) := \text{evolve}^*(E,\ \text{decide}(E, q)).$$

Qui evolve* denota l'applicazione iterata di `evolve` alla lista di eventi risultante. Se il comando è locale a un trial già esistente, si può usare anche la vista:

$$\delta_q^T(T) := \text{evolve}^*(T,\ \text{decide}(T, q)).$$

Il simbolo δ resta quindi utile, ma solo come scorciatoia derivata del canone event-first.

Famiglie del passo. Nel seguito useremo il termine *passo canonico* soprattutto per riferirci alle famiglie `create`, `derive`, `train`, `proc` e `obs` come famiglie di cambiamento del trial espresse da comandi e realizzate da eventi di dominio.

Il confronto rispetto a una vista come operazione osservazionale

I confronti dipendono da *che cosa* si osserva. Introduciamo quindi una vista come proiezione read-only:

$$V : \mathcal{T} \to Y_V.$$

Il *confronto rispetto a una vista* è definito come:

$$\Delta_V(T_1, T_2) := \text{diff}_V\Big(V(T_1),\ V(T_2)\Big).$$

La conseguenza importante è che non esiste un unico confronto assoluto: la differenza osservabile dipende dalla proiezione scelta dal ricercatore. Viste tipiche sono, per esempio, $V_A(T) = A$, $V_H(T) = H$, $V_P(T) = P$, $V_O(T) = O$, $V_L(T) = L$, $V_{\text{core}}(T) = \text{core}(T)$ e $V_M(T) = M(T)$.

Equivalenza sotto vista. Una vista induce anche una nozione di equivalenza osservazionale:

$$T_1 \equiv_V T_2 \iff \Delta_V(T_1, T_2) = \varnothing_V.$$

Due trial possono quindi risultare equivalenti sotto una vista e distinti sotto un'altra. Questo è particolarmente utile quando si confrontano varianti che condividono la stessa architettura ma differiscono, per esempio, nel processo di training, nella genealogia o nelle osservazioni registrate.

Confronto locale come caso particolare. Il confronto tra due stati consecutivi della stessa storia è un caso particolare del livello osservazionale. Se $T_k^{(t)}$ e $T_{k+1}^{(t)}$ sono due stati consecutivi di uno stesso trial ricostruiti via replay, il confronto locale rispetto alla vista V è:

$$\Delta^{\text{step}}_{V,t,k\to k+1} := \Delta_V(T_k^{(t)},\ T_{k+1}^{(t)}).$$

Questa nozione risponde, per esempio, a una domanda semplice ma cruciale: capire che cosa sia cambiato sotto una certa vista in seguito a quel passo, senza confondere il passo canonico con il risultato del confronto.

Correttezza del confronto. Quando un confronto rispetto a una vista è rappresentato come patch o edit script e si dispone di un operatore `apply`, richiediamo la proprietà:

$$\text{apply}(\Delta_V(T_1, T_2),\ V(T_1)) = V(T_2).$$

Nel seguito adottiamo una nozione verificabile di soundness, senza richiedere unicità o minimalità della patch prodotta.

5.4 Lettura operazionale delle trasformazioni architetturali

Quando la componente architetturale A è rappresentata come bigrafo, vale a dire come struttura che combina contenimento e connettività, molte trasformazioni strutturali possono essere lette come passi di riscrittura nei *Bigraphical Reactive Systems* (BRS). Questa lettura operazionale è utile, ma non sostituisce la semantica primaria event-first del framework.

Reaction rule, patch ed evento canonico. Assumiamo una reaction rule $r : L \to R$ applicabile tramite un match o embedding m. La lettura più utile, nel quadro attuale, non è quella di un mutatore alternativo della semantica, ma una pipeline del tipo:

$$p = \text{compileToValidatedPatch}(r, m, A) \qquad A' = \text{apply}(p, A).$$

La reaction rule spiega quindi *perché* e *come* sia stato costruito un certo payload tecnico; la patch descrive *che cosa* viene applicato all'IR; l'evento canonico registra *che cosa entra nella storia*.

Single-mutator invariant. Per la componente architetturale assumiamo un principio semplice ma importante: la mutazione canonica di A passa attraverso un unico mutatore di dominio, espresso nella storia come evento di patch applicata. Eventuali eventi di reaction o di provenance possono documentare la sorgente della trasformazione, ma non devono mutare direttamente l'architettura. In forma concettuale, la provenance può accompagnare la mutazione; non la sostituisce.

Mutazione canonica architetturale. Nel testo della tesi chiameremo quindi *mutazione canonica architetturale* il fatto semantico per cui un trial cambia nella propria componente A lungo la storia canonica. La patch resta il payload tecnico di tale cambiamento; la lettura BRS tramite reaction rules ne fornisce una spiegazione operazionale locale. Questi tre livelli non vanno confusi.

Correttezza strutturale. Le trasformazioni ammissibili devono preservare condizioni di well-formedness del bigrafo e dei vincoli che codificano la semantica di flusso. Una trasformazione che produce una configurazione strutturalmente invalida deve quindi essere rifiutata o segnalata. Questi controlli rientrano negli oracoli di validazione discussi in sezione 5.7.

5.5 Derivazione di trial e lineage

Nel quadro corrente, il concetto principale non è più il *fork* come nome canonico dell'operazione, ma la *derivazione di un nuovo trial*. Il termine *fork* può restare come intuizione informale di branching o come etichetta tecnica locale, ma la lettura concettuale da privilegiare è genealogica.

Derivare un nuovo trial. Una derivazione crea un trial figlio con nuova identità e al più un solo parent. In forma compatta, possiamo scrivere:

$$T \rightsquigarrow_{\rho,\omega} T'$$

dove ρ rappresenta la ragione o trasformazione caratteristica della derivazione e ω la weight policy associata. Nel canone corrente le policy principali sono `reset`, `preserve` e `smooth`. In particolare, `reset` indica assenza di riuso sostanziale dei pesi del parent, `preserve` indica il riuso delle componenti congruenti con inizializzazione del resto, mentre `smooth` è una policy canonica ma oggi supporta soltanto una strategia semplice di blending controllato, vale a dire una transizione graduale tra configurazioni, sufficiente per esprimere l'idea di continuità della transizione senza farne dipendere la semantica generale.

Record minimo di lineage. Il *lineage*, ossia il grafo di derivazione minimo del trial, è già catturato da:

$$L = \langle id, parent, src \rangle.$$

Questa forma è intenzionalmente minima: regola, match, checkpoint sorgente o altri dettagli ricchi possono essere registrati in osservazioni o ricostruiti dalla storia, ma non appartengono al nucleo obbligatorio di L.

Invariante di derivazione. La derivazione è verificabile se parent e child risultano coerenti sul punto di derivazione dichiarato: il child deve dichiarare il parent corretto e la sorgente immediata rilevante, e tale relazione deve risultare compatibile con la storia canonica disponibile. Una certificazione più forte basata su identificatori tecnici o su ancore più ricche può essere utile, ma non appartiene al requisito semantico minimo fissato qui.

Ruolo delle weight policies. Le weight policies non introducono una seconda semantica concorrente del trial: specificano piuttosto come il child si relazioni al parent sul piano dello stato parametrico quando la derivazione crea un nuovo trial. Nel quadro attuale del framework, le classi canoniche rilevanti per la tesi sono `reset`, `preserve` e `smooth`; il loro ruolo qui resta quello di chiarire il significato della derivazione, non di anticipare dettagli implementativi di backend o checkpoint.

Lineage come struttura del dominio. La genealogia dei trial non è soltanto un'informazione di reportistica. Essa fa parte dello stato del trial e rende interpretabili confronti del tipo baseline vs variante, ablation vs incremento, oppure rami generati da controller diversi ma riconducibili a una baseline comune.

5.6 Il *boundary* come contratto verificabile

L'integrazione con strumenti esterni è descritta come un'*interfaccia verso l'esterno* (*boundary*) verificabile, non come un semplice dettaglio infrastrutturale. La formulazione corretta, nello stato attuale del progetto, è *lens-inspired*: l'export produce una vista esterna e la progettazione prevede un complement per preservare l'informazione non rappresentata. Nel caso di studio principale, questo schema astratto prende forma soprattutto nel boundary verso BigraphER, che funge da vista esterna e da strumento di controllo o trasformazione strutturale mantenendo però il framework semanticamente event-first.

Interfacce astratte desiderate. La forma astratta di riferimento resta:

$$\texttt{get} : T \to (V, C) \qquad \texttt{put} : (T, V', C) \to T'.$$

Ciò che conta qui è il lessico del roundtrip — *input / complement / output* — e la possibilità di formulare una nozione controllabile di coerenza tra stato interno, vista esterna e stato rientrato.

Perimetro attuale del boundary. Nel prototipo attuale, il punto ormai chiuso non è una realizzazione piena di *lens with complements*, ma una forma *lens-inspired* sufficiente a rendere verificabili export, rientro e coerenza del roundtrip senza spostare la fonte di verità fuori dal replay. In altre parole, il capitolo fissa qui il contratto semantico del boundary e non richiede di assumere già stabilizzata una componente *complement* first-class in senso forte.

Boundary e replay. L'attraversamento del boundary deve restare compatibile con il replay: lo strumento esterno non diventa fonte di verità del trial, ma attraversa un contratto osservabile e verificabile. In particolare, operazioni come training, export o controlli esterni producono artefatti e provenance che vengono ricondotti a O, mentre il significato canonico continua a essere definito dal replay degli eventi del dominio.

Ruolo del boundary. Il boundary non è quindi un semplice adattatore tecnico: cambia il modo in cui una trasformazione viene giustificata, verificata e reintegrata nella storia del trial. Tuttavia, anche in questo caso il significato canonico resta quello del replay; il boundary aggiunge un contratto di coerenza tra stato interno, vista esterna, complement previsto e output reimportato, senza sostituirsi alla semantica del core.

5.7 Proprietà verificabili e oracoli

In questa tesi, con *proprietà formali* si intendono proprietà definite in modo preciso e verificabili tramite replay, controlli di coerenza e test, non dimostrazioni in un proof assistant. Quando utile, tali proprietà possono essere controllate anche tramite *property-based testing*, secondo l'approccio inaugurato da QuickCheck [4] e disponibile in .NET/F# tramite FsCheck [9].

Proprietà centrali.

- **Correttezza del replay.** Dato lo stesso log di eventi nello stesso ordine e gli stessi artefatti referenziati, il replay ricostruisce lo stesso stato finale e le stesse proiezioni rigenerabili.
- **Unicità del mutatore architetturale.** La componente A cambia solo tramite l'evento canonico che applica la patch architetturale; eventuali eventi di reaction o provenance non possono mutare direttamente l'architettura.
- **Correttezza del confronto.** Per ogni coppia di trial confrontati sotto una vista, Δ_V è coerente con la proiezione scelta; nel caso step-by-step, il confronto locale descrive correttamente la variazione osservabile indotta dal passo canonico corrispondente.
- **Coerenza della derivazione.** Un trial derivato deve poter essere ricondotto in modo controllabile al proprio parent e al proprio punto di derivazione, secondo il lineage minimo dichiarato e la storia canonica disponibile.
- **Coerenza del boundary.** Ogni roundtrip esterno è verificabile: stato di partenza, vista esterna, complement previsto e stato rientrato devono risultare coerenti rispetto al contratto dichiarato.
- **Intercambiabilità dei controller.** Cambiare sorgente dei comandi (gestione manuale vs controller automatico) modifica la provenienza, non la semantica del dominio: a parità di eventi canonici, il replay produce lo stesso stato.

Questi enunciati non sono pensati come principi isolati: correttezza del replay, single-mutator invariant, coerenza della derivazione e coerenza del boundary sono direttamente coerenti con i requisiti non negoziabili del progetto e vengono poi tradotti in oracoli osservabili nel capitolo di valutazione.

Dagli enunciati agli oracoli. Le proprietà precedenti non restano soltanto affermazioni teoriche: vengono tradotte in oracoli controllabili nel capitolo di valutazione. In particolare, il passaggio da proprietà a test segue una logica uniforme: si ricostruisce lo stato tramite replay, si calcolano le proiezioni rilevanti e si verifica che le condizioni dichiarate risultino soddisfatte.

Oracoli di verifica. Le proprietà precedenti possono essere controllate tramite: (i) replay completo e confronto di snapshot o proiezioni rigenerabili; (ii) rigenerazione di Δ_V come read model e controlli di coerenza sulle proiezioni; (iii) verifica che le mutazioni architetturali osservate corrispondano all'evento canonico di patch e non a eventi di provenance; (iv) controllo della coerenza lineage tra parent e child rispetto alla storia disponibile; (v) verifica della coerenza del roundtrip di boundary; (vi) comparazione di storie canoniche equivalenti con metadati di provenienza differenti.

Chiusura del capitolo. Il risultato di questo capitolo può essere riassunto così: il framework resta trial-first ed event-first, il replay ne fissa la fonte di verità, il passo canonico ne compatta alcune famiglie di cambiamento, il confronto rispetto a una vista ne legge le differenze osservabili e il boundary ne disciplina gli attraversamenti esterni. Su questa base il capitolo di workflow potrà mostrare come il framework viene usato nella pratica, mentre il capitolo di valutazione tradurrà queste stesse nozioni in oracoli osservabili.

Capitolo 6

Architettura del framework

Questo capitolo descrive come la semantica definita in Capitolo 5 prende forma in un'architettura software concreta. Il framework adotta un'impostazione coerente con lo stile *Domain Modeling Made Functional* (DMMF): il dominio costituisce un core deterministico, mentre training, I/O e integrazioni con tool esterni sono confinati oltre un'interfaccia verso l'esterno (*boundary*) degli effetti. [8], [28] In questo modo, le decisioni sulla struttura del trial restano separate dalle tecnologie usate per eseguire training, persistenza ed esportazioni. La notazione di riferimento è fissata in Appendice A: il trial completo è descritto come $T = \langle A, H, W, P, O, L \rangle$, con $core(T) = \langle A, H, W \rangle$, mentre il modello in senso stretto può essere letto come vista derivata $M(T) = \langle A, W \rangle$. Nel seguito, il capitolo usa questa notazione per mostrare come il significato del trial venga preservato mentre l'architettura software organizza persistenza, orchestrazione ed effetti esterni.

6.1 Obiettivo architetturale

L'obiettivo architetturale è tradurre in componenti software le proprietà già motivate nei capitoli precedenti: tracciabilità della ricerca, riproducibilità del trial, separazione tra semantica di dominio ed effetti esterni, e possibilità di integrare backend e tool diversi senza riscrivere il core. Più concretamente, il framework deve garantire che:

- il log degli eventi resti la fonte di verità del trial;
- lo stato sia ricostruibile tramite replay deterministico;
- training, esportazioni e verifiche esterne siano trattati come effetti orchestrati e tracciati;
- viste, report e confronti restino proiezioni rigenerabili, non parte del core di dominio.

Questa scelta architetturale permette di mantenere stabile il significato dei cambiamenti anche quando cambiano backend, formati di esportazione o strumenti di supporto. Letta rispetto ai requisiti non negoziabili del progetto, essa concretizza in componenti software tre vincoli precisi: bigraph-first per la componente architetturale A, vale a dire basato su una rappresentazione architetturale esprimibile

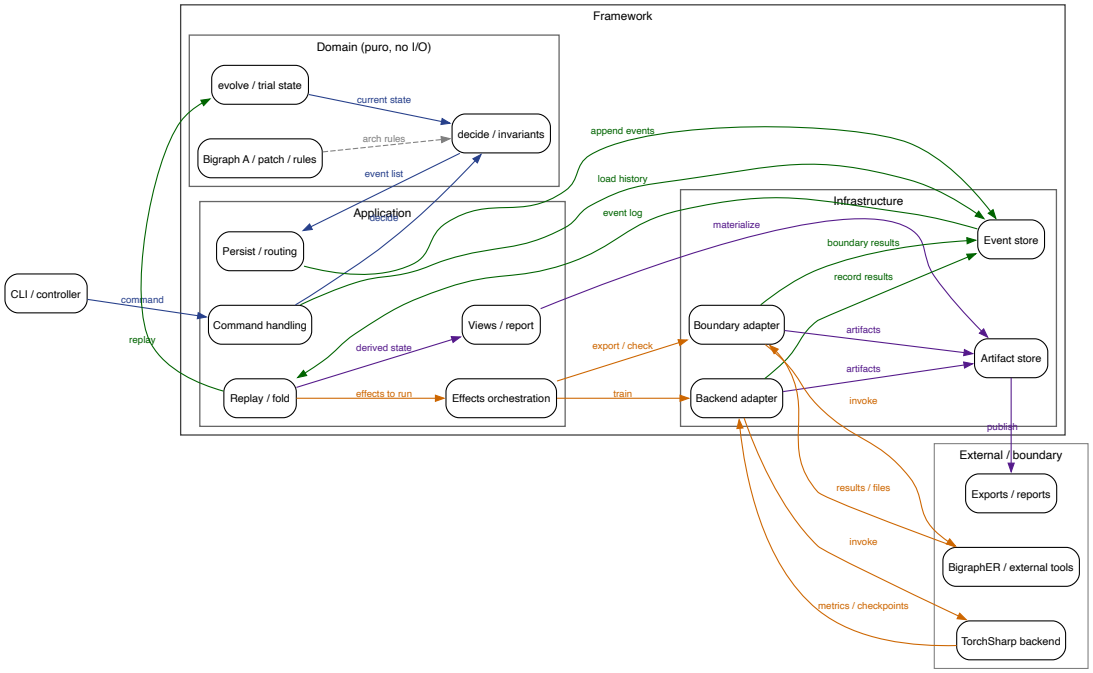


Figura 6.1: Architettura end-to-end del framework: core deterministico a eventi (decide/evolve/replay) e interfaccia verso l'esterno (*boundary*) degli effetti, con backend e tool esterni orchestrati e tracciati nel log.

tramite bigrafi, replay deterministico come fonte di verità del core, e interfaccia verso l'esterno (*boundary*) esplicita per training e tool esterni in Application e Infrastructure. La Figura 6.1 riassume il flusso end-to-end e rende esplicito il confine tra core deterministico ed effetti esterni.

Questa impostazione si appoggia anche a due principi di design coerenti con il core a eventi del framework. Il primo è *make illegal states unrepresentable*: il Domain deve esprimere tipi, invarianti e validazioni in modo da rendere non ammissibili, già al livello del modello, configurazioni che violerebbero la struttura del trial o dei suoi componenti. Il secondo è *fail fast*: un comando che produrrebbe uno stato incoerente deve essere rifiutato il prima possibile da `decide` o dai controlli di validazione, invece di propagare inconsistenze verso backend, tool esterni o read models. Architetturalmente, questi principi traducono in struttura software requisiti già fissati dal progetto: replay deterministico del core, validazione delle trasformazioni strutturali e separazione tra significato del dominio ed effetti orchestrati.

6.2 Flusso end-to-end

Dal punto di vista architetturale, il framework realizza un ciclo regolare: un controller propone un comando, il dominio decide se il comando è ammissibile, il log registra gli eventi prodotti, e l'application layer orchestra gli eventuali effetti esterni. Il risultato è una traiettoria interamente ricostruibile a partire dalla storia del trial.

Passi principali.

1. **Emissione del comando.** Un controller manuale o automatico produce un `Command` riferito a uno specifico trial.

2. **Decisione di dominio.** Il core applica `decide` sullo stato corrente e produce una lista di `Event`, oppure rifiuta il comando.

3. **Persistenza.** Gli eventi vengono aggiunti al log append-only che costituisce la storia canonica del trial.

4. **Evoluzione dello stato.** Il replay/fold degli eventi aggiorna lo stato canonico del trial.

5. **Orchestrazione degli effetti.** Se tra gli eventi compaiono richieste di lavoro esterno, l'application layer attiva backend o tool appropriati.

6. **Proiezioni.** Read models, report, timeline e visualizzazioni vengono aggiornati o rigenerati dal log.

Perche' questo flusso conta. Architetturalmente, il punto chiave non è soltanto la sequenza dei passi, ma il fatto che il dominio non esegue direttamente effetti irreversibili. Il core decide e registra; l'esterno esegue e restituisce risultati tracciati. Questa separazione rende il sistema più verificabile, più semplice da testare e meno dipendente da una specifica toolchain. Rende inoltre concreta, a livello di architettura, la triade di requisiti non negoziabili del progetto: event sourcing del core, training come effetto registrato e interfaccia verso l'esterno (*boundary*) esplicita verso strumenti esterni.

6.3 Layering: Domain, Application, Infrastructure

L'architettura è organizzata in tre livelli principali, con dipendenze orientate verso il dominio. Questa separazione non è solo una scelta implementativa: è il modo con cui la semantica della tesi viene protetta dall'accoppiamento con backend, librerie e formati esterni.

Domain. Il *Domain* contiene il modello di stato del trial, i tipi `Command` e `Event`, la funzione `decide` e la funzione `evolve`. Qui risiedono gli invarianti del modello e le regole che stabiliscono quali cambiamenti sono ammissibili. Il Domain non effettua I/O e non dipende da backend specifici. In particolare, è qui che il framework resta bigraph-first per la sola componente architetturale A, mentre le altre componenti del trial seguono il rispettivo dominio canonico senza essere forzate dentro una rappresentazione bigrafica unica. La scelta bigraph-first riguarda quindi la sola architettura del modello, non l'intero stato del trial.

Application. L'*Application* orchestra i workflow. Riceve i comandi, invoca il Domain, gestisce la persistenza degli eventi, coordina backend e interfacce verso l'esterno (*boundary*), e costruisce o aggiorna le proiezioni necessarie per l'osservazione. È anche il livello in cui richieste e completamenti di effetti esterni vengono ordinati in un flusso coerente con la storia del trial, senza spostare nel dominio responsabilità che appartengono all'orchestrazione.

Infrastructure. L'*Infrastructure* realizza le dipendenze esterne: persistenza del log, artifact store, backend di training, esportatori/importatori, integrazioni con tool esterni e servizi di visualizzazione. A questo livello cambiano i dettagli tecnologici; la semantica del trial, invece, resta invariata.

Beneficio della separazione. Questa stratificazione permette, ad esempio, di sostituire il backend di training, aggiungere un nuovo esportatore, o introdurre un adapter verso un tool esterno senza modificare la logica di dominio. Di conseguenza, il framework resta estendibile senza rinunciare alla tracciabilità della storia sperimentale.

6.4 Controller e orchestrazione dei comandi

Il controller è la sorgente dei comandi della ricerca. Nel caso più semplice coincide con il ricercatore che opera manualmente, un comando alla volta; in prospettiva può essere sostituito o affiancato da un agente automatico. Dal punto di vista architetturale, però, entrambi condividono lo stesso contratto: producono `Command` e metadati di provenienza, ma non modificano direttamente lo stato del dominio.

Intercambiabilità. L'intercambiabilità tra controller manuale e controller automatico deriva dal fatto che la decisione di dominio è funzione dello stato e del comando, non dell'identità del controller. Le informazioni su autore, seed, versione dell'agente o contesto di esecuzione restano metadati della richiesta applicativa o dell'evento registrato. Questo permette di cambiare strategia di ricerca senza cambiare il significato degli eventi già registrati.

6.5 Backend ed effetti esterni

Le operazioni non pure, come training, valutazione, esportazione o invocazione di tool esterni, non appartengono al Domain. Sono invece trattate come effetti esterni orchestrati da Application e realizzati da Infrastructure. Questa scelta è coerente con la semantica presentata nel capitolo precedente: il log registra richieste e risultati, mentre l'esecuzione concreta avviene oltre l'interfaccia verso l'esterno (*boundary*).

Training come richiesta tracciata. Il training viene rappresentato come una richiesta di lavoro esterno, seguita da eventi che registrano metriche, checkpoint e completamento dell'operazione. Il vantaggio architetturale è duplice: si evita di

incorporare il backend nel Domain, e si mantiene la storia del trial leggibile anche quando il training richiede tempo, fallisce o viene ripetuto.

Backend sostituibili. Il prototipo attuale usa TorchSharp, un binding .NET per PyTorch, per eseguire training e valutazione, ma l'architettura è costruita in modo da separare il significato del trial dal meccanismo numerico che lo esegue. Il backend, in altri termini, interpreta una rappresentazione eseguibile derivata dal trial ma non ridefinisce il significato degli eventi. [25] Questa separazione consente di ragionare in termini di backend sostituibili senza trasformare il capitolo architetturale in un catalogo di dettagli implementativi.

6.6 Boundary e integrazione di tool esterni

Oltre ai backend di training, il framework può dialogare con tool esterni dedicati a trasformazioni, analisi o verifiche. Architetturalmente, queste interazioni sono confinate in un'interfaccia verso l'esterno (*boundary*) esplicita: il trial interno viene esportato in una rappresentazione esterna, il tool opera su tale rappresentazione, e l'eventuale rientro viene riconciliato tramite import controllato.

Roundtrip come contratto architetturale. Un roundtrip può essere descritto come

$$T \xrightarrow{\text{export}} X \xrightarrow{\text{import}} T'.$$

Qui X denota un artefatto o una rappresentazione esterna, per esempio un file `.big`, un file `.dot` o una vista serializzata del trial. L'architettura deve rendere espliciti: (i) quale vista del trial viene esportata, (ii) quali informazioni di contesto devono accompagnare tale vista, (iii) quali controlli vengono eseguiti al rientro. Questo evita che l'integrazione con tool esterni introduca trasformazioni opache.

Esempio: BigraphER. BigraphER è un esempio naturale di tool integrabile nell'interfaccia verso l'esterno (*boundary*): consente di enumerare o applicare riscritture sui bigrafi, ma tali operazioni restano esterne al core. Nel quadro del framework, questo significa che una trasformazione ottenuta su una vista bigrafica della componente architetturale A non diventa automaticamente storia canonica del trial: il risultato deve essere riconciliato con il canone a eventi tramite import controllato e corrispondenti cambiamenti di dominio. Il framework incapsula quindi export, invocazione del tool, raccolta dei risultati e import, mantenendo tracciabilità e verificabilità del roundtrip. [21]

Views pipeline ed esportazioni. Anche esportazioni unidirezionali, come DOT o report strutturali, seguono la stessa logica: sono artefatti derivati da viste del trial, utili per osservazione e interoperabilità, ma non ridefiniscono lo stato canonico.

6.7 Artifact store e read models

Artefatti immutabili. Checkpoint, report, esportazioni e file generati dai tool esterni sono trattati come artefatti immutabili referenziati dal log. Questa scelta evita di appesantire il log con dati voluminosi e consente di mantenere separati storia canonica e output pesanti. Sul piano architetturale, il principio da preservare è semplice: il log resta la fonte primaria, mentre file, checkpoint e report vivono come prodotti derivati stabili del run.

Read models rigenerabili. Timeline di eventi, curve di metriche, report di confronto e viste strutturali sono ottenuti come proiezioni rigenerabili dal log. Da un punto di vista architetturale, questo significa che l'osservabilità del sistema può evolvere nel tempo senza cambiare il core di dominio: nuove viste possono essere introdotte come ulteriori read models. La distinzione da mantenere esplicita è quindi la seguente: lo stato canonico resta ricostruito via replay, gli artefatti pesanti vivono nell'artifact store, mentre read models e report restano superfici derivate utili a osservazione, confronto e debugging.

Ports e adattatori. Per mantenere esplicite le dipendenze esterne, l'Application programma contro ports minimi, mentre Infrastructure fornisce gli adattatori concreti. È in questo punto del codice che la distinzione tra core e interfaccia verso l'esterno (*boundary*) diventa verificabile anche a livello implementativo. Il dettaglio di questi adattatori appartiene però al capitolo di implementazione: qui interessa soprattutto il fatto che il layering impedisca alle dipendenze tecnologiche di ridefinire la semantica del trial.

6.8 Sintesi

L'architettura del framework realizza in forma software la distinzione, già introdotta sul piano del modello, tra un core deterministico e un insieme di effetti esterni orchestrati. Il layering Domain/Application/Infrastructure, il flusso `Command` → `decide` → `Event` → replay, l'interfaccia verso tool specialistici (*boundary*) e l'uso di artefatti e read models rigenerabili consentono di mantenere stabile il significato del trial pur lasciando aperta l'evoluzione dei backend e delle integrazioni. L'architettura rende così visibile, sul piano software, la stessa separazione che il capitolo precedente ha fissato sul piano semantico: storia canonica del trial da un lato, proiezioni, esecuzioni e integrazioni esterne dall'altro.

Capitolo 7

Workflow e casi d'uso

Questo capitolo descrive come un ricercatore in modalità manuale o un controllore automatico usa il framework nella pratica per condurre esperimenti di *Automated Machine Learning* (AutoML) *tracciabili* e *verificabili*. Il focus è su un flusso di lavoro *sequenziale* e *passo per passo* (*one-by-one*), con UX operativa primaria *command-first*: una modifica alla volta, con training e valutazione intercalati, e con possibilità di derivare varianti a partire da una baseline comune. Questa impostazione rende naturale una politica sperimentale di variazione controllata a un cambiamento per volta, senza trasformare il capitolo in una trattazione metodologica separata.

Nel seguito usiamo la notazione già fissata nell'appendice e nei capitoli precedenti: il contenitore logico dell'attività sperimentale è l'*esperimento* $E = \langle C, \mathcal{T} \rangle$, mentre l'unità evolutiva è il *trial*

$$T = \langle A, H, W, P, O, L \rangle,$$

dove A è l'architettura, H i parametri di configurazione rilevanti, W i pesi, P lo stato di processo, O le osservazioni e L il *lineage minimo*, ossia il grafo di derivazione minimo del trial. Quando utile, useremo anche le proiezioni

$$\mathrm{core}(T) = \langle A, H, W \rangle, \qquad M(T) = \langle A, W \rangle.$$

I dettagli semantici del replay, del confronto sotto vista e dell'interfaccia verso l'esterno (*boundary*) sono definiti nei capitoli precedenti; qui il punto di vista è volutamente operativo: *che cosa fa il ricercatore*, *in quale ordine*, e *quali artefatti osserva*. Le figure e i casi d'uso del capitolo hanno quindi lo scopo di mostrare il workflow del framework, non di ridefinirne la semantica o di anticiparne il mapping implementativo. [8], [18] Quando serve invece una lettura completa di pacchetti sperimentali e artefatti maturi, il rimando naturale è a Capitolo 10.

Gerarchia delle superfici operative. Nel quadro attuale del framework, la DSL canonica del workflow resta lo script di sessione `session.jsonl`. CLI, script modulari e notebook sono superfici operative che aiutano a comporre, eseguire, ispezionare o documentare il workflow, ma non spostano la fonte di verità fuori dalla storia degli eventi e dal replay. In particolare, il notebook resta un layer di orchestrazione, analisi ed evidenza: utile per leggere pacchetti sperimentali e artefatti, ma non sorgente primaria della logica canonica. Una UX in stile *Bigraphical Reactive Systems* (BRS) resta invece una modalità avanzata per proporre o leggere

trasformazioni strutturali tramite regole e match, senza sostituire il livello operativo command-first. In prospettiva, superfici più interattive come `ViewAction` o report di `enabled_actions` possono arricchire il workflow, ma restano anch'esse viste derivate e non la DSL canonica del sistema.

7.1 Vista end-to-end del workflow sperimentale

Idea generale. L'attività sperimentale può essere letta come un ciclo unico che parte da un esperimento, introduce un primo trial, ne deriva varianti, le allena, ne registra le osservazioni e le confronta sotto viste utili alla ricerca. In forma compatta, il ciclo è

$$\texttt{create} \to \texttt{derive} \to \texttt{train} \to \texttt{proc/obs} \to \texttt{compare} \to \texttt{derive} \to \cdots$$

e va inteso come istanziazione operativa della grammatica basata su *event sourcing* già discussa nei capitoli precedenti.

Ruolo di esperimento e trial. L'esperimento $E = \langle C, \mathcal{T} \rangle$ fornisce il contesto stabile dell'attività sperimentale e raccoglie lo spazio dei trial generati. Il trial $T = \langle A, H, W, P, O, L \rangle$ è invece l'unità concreta su cui il ricercatore agisce: l'architettura A può essere trasformata, i pesi W possono essere aggiornati dal training, lo stato di processo P scandisce le fasi del workflow, le osservazioni O registrano metriche e altri risultati, mentre L conserva il *lineage minimo* che collega il trial alla baseline da cui deriva.

Lettura pratica del ciclo. Un uso tipico del framework segue cinque momenti ricorrenti:

1. **Create.** Si introduce un trial iniziale T_0 a partire da una configurazione base dell'esperimento.
2. **Derive.** Si genera una variante T_1 modificando una parte del trial di partenza, per esempio l'architettura o gli iperparametri.
3. **Train.** Si esegue training sul trial corrente, aggiornando soprattutto W e facendo avanzare P.
4. **Obs/Proc.** Si registrano metriche, checkpoint, fasi e altra evidenza osservabile in O.
5. **Compare.** Si confrontano trial diversi sotto viste appropriate, per poi decidere se continuare il training, fermarsi o derivare una nuova variante.

Perchè questa vista è utile. Questa lettura end-to-end unifica i tre casi d'uso del capitolo: UC1 è una derivazione interna ottenuta tramite trasformazione architetturale; UC2 è una derivazione o modifica mediata da uno strumento esterno; UC3 mostra come più trial derivati da una stessa baseline restino confrontabili nel

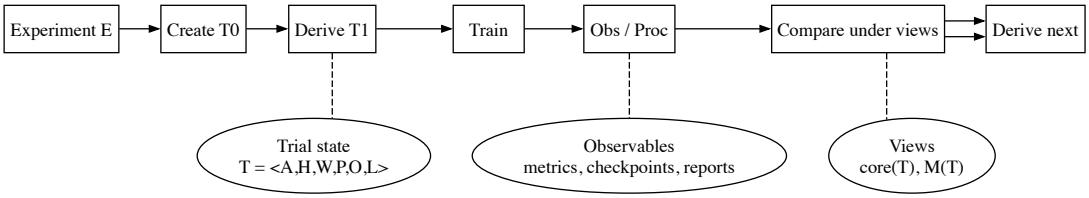


Figura 7.1: Vista end-to-end del workflow sperimentale: un esperimento organizza trial che vengono creati, derivati, allenati, osservati e confrontati sotto viste appropriate, producendo artefatti leggibili lungo tutta la traiettoria.

tempo. In tutti i casi, il ricercatore osserva non solo il risultato finale, ma anche la traiettoria che lo ha prodotto. La Figura 7.1 sintetizza questa lettura operativa in forma compatta.

7.2 Workflow sequenziale one-by-one

Passi tipici del loop di ricerca. Dal punto di vista del ricercatore, il flusso di lavoro tipico è il seguente:

1. **Inizializza un trial** con una configurazione di base: architettura A_0, iperparametri H_0, stato di processo iniziale P_0 e nessun peso o un checkpoint di partenza.
2. **Allena per N epoche** tramite un backend esterno, producendo metriche e, quando opportuno, checkpoint; il trial aggiorna soprattutto W, P e O.
3. **Applica una modifica** come trasformazione discreta e tracciabile:
   - mutazione architetturale interna su A (UC1, sezione 7.3), [13]
   - roundtrip verso uno strumento esterno (UC2, sezione 7.4), [7]
   - aggiornamento di iperparametri o configurazione di processo.
4. **Allena ancora per M epoche** secondo una politica esplicita di prosecuzione (per esempio riavvio, riuso selettivo o transizione graduale dei pesi, a seconda della natura della modifica).
5. **Deriva varianti opzionali** per confrontare alternative a partire da una baseline comune (UC3, sezione 7.5).
6. **Analizza e confronta** usando il *lineage*, viste derivate, metriche annotate e visualizzazioni dell'architettura.

Output osservabili. Un workflow completo produce tipicamente: (i) una storia di eventi da cui il trial corrente può essere ricostruito, (ii) checkpoint, metriche e altri artefatti di training, (iii) confronti rispetto a viste rilevanti per la domanda sperimentale, (iv) eventuali export e re-import verso tool esterni, (v) visualizzazioni dell'architettura e del grafo di derivazione. La scelta di trattare checkpoint, metriche

e report come artefatti derivati e referenziabili è coerente con la prassi dei sistemi di tracciamento sperimentale, in cui metadati e artefatti vengono mantenuti distinti ma collegati. [29]

7.3 UC1: Mutazione interna e confronto architetturale

Obiettivo. Eseguire una modifica architetturale *una alla volta* senza ricorrere a flag o configurazioni divergenti nel codice: la variante nasce da una trasformazione locale dell'architettura A e viene poi osservata tramite un confronto rispetto alla vista architetturale. [13]

Scenario. Si parte da un trial $T_k = \langle A_k, H_k, W_k, P_k, O_k, L_k \rangle$. Il ricercatore seleziona una trasformazione interna — per esempio inserire un blocco, sostituire un modulo o rimuovere un componente — e la applica al trial corrente. Se l'architettura è rappresentata tramite una rappresentazione intermedia (IR) basata su bigrafo, la modifica può essere letta come una riscrittura locale $A_k \to A_{k+1}$, ma dal punto di vista del workflow ciò che conta è più semplice: il trial risultante espone una nuova architettura osservabile e rimane confrontabile con il trial precedente.

Esempi di trasformazioni locali. Tre casi tipici sono:

- **sostituzione,** ad esempio di un blocco convoluzionale 3×3 con uno 5×5 a interfaccia preservata;
- **inserimento,** ad esempio di un nuovo blocco tra due componenti già connesse;
- **rimozione,** ad esempio di un modulo intermedio ridondante o di un componente di regolarizzazione che si vuole ablare.

Questi esempi sono sufficienti a rendere chiara la lettura *one change at a time*: a ogni passo, il ricercatore vuole sapere *che cosa è cambiato* nell'architettura e *che effetto ha prodotto* dopo ulteriore training.

Per chiudere UC1 con evidenza riproducibile, nel prototipo si possono usare pacchetti sperimentali che rendono visibile il punto di mutazione, il confronto tra baseline e trial derivato e una sintesi finale delle metriche osservate.

Output osservabili. Dopo la trasformazione, il ricercatore osserva tipicamente:

- **confronto architetturale sotto vista,** per evidenziare aggiunte, rimozioni o sostituzioni nella struttura di A;
- **visualizzazioni,** per esempio un export DOT della topologia o una vista più fedele all'IR architetturale;
- **metriche nel tempo,** ottenute dopo ulteriore training, così da collegare il punto di cambiamento strutturale all'andamento osservato di loss e accuratezza.

Lettura del caso d'uso. UC1 mostra quindi il cuore del workflow interno: modificare il trial, proseguire il training e leggere il risultato sotto una vista architetturale che renda esplicita la differenza tra *prima* e *dopo*.

Nel prototipo questa lettura è già coperta da pacchetti sperimentali in cui una baseline e un trial derivato con una sola mutazione interna restano leggibili tramite viste strutturali, metriche e grafo di derivazione. Qui basta fissare l'aggancio operativo tra UC1 e queste superfici osservabili; la discussione di un caso sperimentale completo viene ripresa in Capitolo 10, mentre il capitolo successivo usa scenari più leggeri per la verifica property-driven.

7.4 UC2: Roundtrip di boundary e reintegrazione verificabile

Obiettivo. Integrare strumenti esterni mantenendo continuità del workflow, tracciabilità e verificabilità del risultato: lo strumento esterno partecipa al processo sperimentale, ma il trial risultante resta leggibile e confrontabile all'interno del framework. [7], [21]

Scenario. Dato uno stato T_k con architettura A_k, il ricercatore richiede una rappresentazione esportabile della parte rilevante del trial, usa uno strumento esterno per esplorare o applicare una trasformazione, e infine reintegra il risultato nel framework. In forma astratta, il roundtrip si articola in quattro passi:

1. **Export:** il framework produce una vista esterna adatta allo strumento e conserva le informazioni necessarie per reinterpretarla correttamente al rientro;
2. **List/Explore:** lo strumento esterno espone transizioni o trasformazioni candidate;
3. **Apply:** il ricercatore seleziona una trasformazione e ottiene una nuova vista esterna;
4. **Import:** il framework reintegra la vista risultante, controlla la coerenza del rientro e produce un nuovo stato confrontabile con quello di partenza.

La Figura 7.2 riassume questi passi come workflow continuo, rendendo esplicita la separazione tra trasformazione esterna e reintegrazione verificabile.

Output osservabili. In questo caso d'uso sono particolarmente importanti:

- **la traccia del roundtrip,** che collega strumento, vista esportata, trasformazione scelta ed esito del rientro;
- **il confronto prima/dopo,** tipicamente sotto la vista architetturale, per capire che cosa è cambiato dopo il passaggio esterno;
- **la riproducibilità,** perchè il risultato accettato del roundtrip entra nella storia del trial e può quindi essere ricostruito senza rieseguire lo strumento esterno.

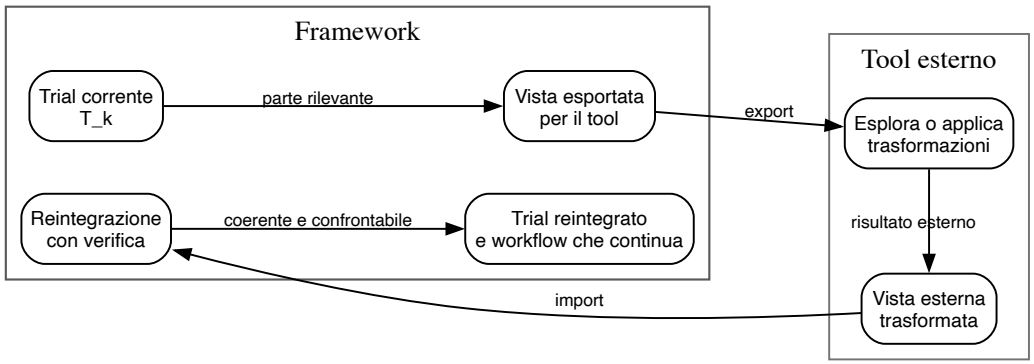


Figura 7.2: UC2: roundtrip di boundary come workflow leggibile e verificabile. Il trial corrente viene esportato sotto una vista adatta allo strumento esterno, il risultato trasformato rientra nel framework tramite reintegrazione verificabile e il workflow prosegue su un trial ancora confrontabile con quello di partenza.

Lettura del caso d'uso. UC2 rende visibile che il framework non è un ambiente chiuso. Può attraversare l'interfaccia verso l'esterno (*boundary*), usare uno strumento specializzato e rientrare senza perdere controllo sul *lineage* sperimentale. La confrontabilità del trial resta quindi preservata.

Nel prototipo attuale questa idea è già coperta da tracce curate di roundtrip: un trial sorgente viene esportato sotto una vista eseguibile, riattraversa il *boundary* tramite import e proiezione verificabile, e rientra nel workflow con controlli di coerenza e artefatti osservabili. La lettura resta volutamente ispirata alle *lenses*, cioè a costrutti bidirezionali: la tesi rende esplicito il lessico input/complement/output, ma il complement first-class non è ancora una componente stabilizzata del prototipo.

7.5 UC3: Trial derivati e confronto tra varianti

Obiettivo. Confrontare alternative in modo controllato e tracciabile: più varianti partono dalla stessa baseline, divergono con una modifica per ramo e vengono allenate o valutate separatamente. Il caso d'uso mostra anche che il controllo manuale o automatico è intercambiabile: cambia la provenienza dell'azione, non il significato del workflow.

Scenario. Il ricercatore esegue una fase comune di baseline:

1. introduce un trial radice e lo allena per un certo budget iniziale;
2. opzionalmente fissa una baseline condivisa con una modifica preliminare;
3. deriva due o più trial figli a partire dallo stesso stato di riferimento.

Da questo punto in poi, i trial derivati divergono:

- **Trial A:** applica una modifica interna e prosegue il training;

- **Trial B:** applica una modifica alternativa, eventualmente mediata da uno strumento esterno, e prosegue il training.

Ogni trial produce metriche e artefatti propri, mantenendo però il lineage verso la stessa baseline.

Confronto sotto viste. Il confronto tra trial è parametrico rispetto alla domanda sperimentale:

- per confronti architetturali si osserva la sola struttura di A;
- per confronti di configurazione si osservano i parametri rilevanti di H;
- per confronti tra modelli allenati si considerano insieme progettazione e stato dei pesi tramite la vista di modello $M(T) = \langle A, W \rangle$.

Questo rende naturale l'analisi *one change at a time*: a baseline comune, la differenza osservata è attribuibile con maggiore chiarezza alla singola modifica introdotta in ciascun ramo.

Controllo manuale o automatico. Lo stesso workflow può essere guidato da un ricercatore in modalità manuale o da un controllore automatico. Dal punto di vista del caso d'uso, ciò che cambia è soprattutto la provenienza delle azioni: il trial risultante resta comunque interpretabile con gli stessi strumenti di confronto, di *lineage* e di osservazione.

Lettura del caso d'uso. UC3 mostra quindi come il framework supporti l'esplorazione comparativa di varianti a baseline comune, mantenendo allineati *lineage*, metriche e artefatti prodotti da ciascun ramo.

La Tabella 7.1 riassume i tre casi d'uso UC1–UC3 in termini di obiettivi, operazioni principali e output osservabili.

Caso d'uso	**Obiettivo**	**Operazioni principali**	**Output osservabili**
UC1	Mutazione architetturale interna, una modifica alla volta	Derivazione del trial corrente, trasformazione locale di A, ulteriore training, confronto sotto vista architetturale	Prima/dopo strutturale, visualizzazione dell'architettura, metriche annotate nel tempo
UC2	Roundtrip verso uno strumento esterno con reintegrazione verificabile	Export di una vista o artefatto rilevante, elaborazione esterna, reimport, controllo di coerenza, prosecuzione del workflow	Traccia del roundtrip, differenza osservabile prima/dopo, artefatti esterni reintegrati, trial confrontabile
UC3	Confronto tra trial derivati da una baseline comune	Baseline iniziale, derivazione di più varianti, training separato, confronto sotto viste rilevanti, selezione della variante più utile	Lineage dei trial, metriche per ramo, confronto tra varianti, selezione finale

Tabella 7.1: Sintesi dei principali casi d'uso del workflow sperimentale. I tre casi d'uso mostrano rispettivamente mutazioni interne, roundtrip di boundary e confronto tra trial derivati da una baseline comune.

Capitolo 8

Implementazione prototipale

Questo capitolo descrive il *prototipo attuale* del framework proposto, con tre obiettivi complementari: (i) mostrare come i concetti dei capitoli precedenti prendono corpo in tipi, moduli e responsabilità concrete, (ii) rendere leggibili e riproducibili i workflow UC1–UC3 tramite superfici operative reali, (iii) documentare le scelte implementative che preservano replay, tracciabilità e separazione tra core puro ed effetti esterni.

Il capitolo non ridefinisce la semantica del framework: assume come dati i contratti concettuali già fissati nei capitoli precedenti e mostra come essi vengano incarnati nel codice attuale.

8.1 Mappatura tra concetti della tesi e realizzazione software

Nel lessico concettuale della tesi, il *trial*, ossia l'unità sperimentale primaria — può essere letto come

$$T = \langle A, H, W, P, O, L \rangle, \qquad core(T) = \langle A, H, W \rangle, \qquad M(T) = \langle A, W \rangle.$$

Nel prototipo attuale questi concetti sono realizzati soprattutto dallo stato concreto del trial nel layer Domain, mentre la vista a livello di esperimento viene ottenuta come proiezione a esecuzione sugli stessi stream di trial. La corrispondenza va letta come mappatura concettuale tra tesi e codice, non come equivalenza letterale 1:1 tra nomi matematici e nomi dei tipi F#. La rappresentazione intermedia (IR) dell'architettura coincide, nel prototipo attuale, con la struttura bigrafica usata dal Domain per descrivere la componente A.

- A (architettura) é realizzata come `Bigraph.T` nel Domain; rappresenta la componente strutturale del trial ed é la parte per cui il framework resta centrato su una rappresentazione a bigrafo.[13]
- H (iperparametri) é realizzato nel prototipo come `Map<string,obj>` associata al sottorecord di modello; la scelta é intenzionalmente minimale e può essere raffinata senza cambiare il contratto concettuale.
- W (weights) é realizzato come `CheckpointRef option`: il log non serializza i pesi, ma conserva riferimenti ai checkpoint prodotti dal backend.

- P (stato di processo) é realizzato dal sottorecord `Proc`, che raccoglie stato di training/esecuzione come budget, seed e fase.
- O (osservazioni) é ormai una componente esplicita dello stato del trial: metriche, checkpoint, fasi e provenienza osservazionale non vivono più in un contenitore opaco, ma in una struttura dedicata e ricostruibile via replay.
- L (grafo di derivazione, *lineage*) é una componente esplicita dello stato del trial e rende visibili identità del trial, parent e sorgente della derivazione.

Il codice corrente mantiene come unità operativa reale *uno stream JSONL per trial*, ma offre anche una *vista di esperimento* che proietta quegli stessi stream in una forma più vicina al modello canonico dell'esperimento. In altre parole, il prototipo non introduce un secondo aggregato persistito per l'esperimento: lo ricostruisce per proiezione a partire dagli stream dei trial. Checkpoint, export, report e altri riferimenti ad artefatti appartengono quindi al trial completo soprattutto tramite O, senza diventare parte obbligata del solo core(T) mostrato nella notazione più leggera.

8.2 Struttura implementativa e responsabilità dei moduli

Il codice segue un layering coerente con *Domain Modeling Made Functional* (DMMF), vale a dire uno stile che separa logica di dominio pura, orchestrazione applicativa e infrastruttura.[28] Il layer *Domain* contiene logica pura e tipi del dominio, il layer *Application* orchestra comandi, replay e proiezioni, il layer *Infrastructure* realizza persistenza, backend e integrazioni esterne.

La Tabella 8.1 riassume questa ripartizione in termini di concetti e responsabilità implementative, mettendo in corrispondenza il lessico della tesi con i ruoli principali del prototipo.

Domain (core deterministico)

- Il layer `TesiBRS.Domain` contiene il nucleo puro del prototipo: stato del trial, comandi, eventi, funzioni di `decide` e `evolve`, patch, reaction rules, invarianti e contratto del *boundary*.
- L'architettura del modello é rappresentata come `Bigraph.T`; la semantica del framework resta basata su *event sourcing*, ma la componente A continua a essere trattata tramite bigrafo.
- Le patch costituiscono il payload tecnico delle mutazioni su architettura, iperparametri e processo; la loro validazione appartiene al core, non all'infrastructure.
- Le reaction rules restano nel dominio come meccanismo di trasformazione e provenienza; nel canone implementativo corrente il cambiamento strutturale effettivo passa comunque per `PatchApplied`.

Concetto	Responsabilità implementative (indicative)
Stato del trial e semantica a eventi	tipi di stato, comandi, eventi, decisione e replay nel layer `TesiBRS.Domain`
Vista di un esperimento	proiezione a esecuzione dell'esperimento a partire dagli stream per-trial, con indici operativi per trial, figli e frontiera
IR basata su bigrafo + invarianti	rappresentazione architetturale, registry della signature, validazione strutturale e utilità di esportazione/proiezione
Patch e reaction rules	patch su architettura/hyper/proc, compilazione di reaction rules e validazione del payload mutazionale
Pipeline applicativa	command handler, routing delle derivazioni, replay, proiezioni e tabelle derivate
Persistenza eventi/artefatti	event store append-only in JSONL e riferimenti ad artefatti esterni come checkpoint e report
Backend e interfaccia verso l'esterno (*boundary*)	backend fake e TorchSharp, adattatori per BigraphER e altri passaggi verso strumenti esterni
CLI, notebook e script	superfici operative per il ciclo di vita dei trial, training, viste, export e casi d'uso riproducibili

Tabella 8.1: Mappatura tra concetti della tesi e ruoli principali del prototipo attuale.

- Il modulo `Experiment` introduce una vista a livello di esperimento coerente con il lessico della tesi, senza duplicare la persistenza rispetto agli stream dei trial.

Application (orchestrazione e read model)

- Il *command handler* applicativo realizza la pipeline concreta `load` → `fold` → `decide` → `append`, isolando l'I/O e lasciando puro il Domain.
- Lo stesso layer realizza il routing delle derivazioni tra stream parent/child e ricostruisce la proiezione a livello di esperimento a partire dagli stream JSONL.
- Moduli di viste e tabelle derivate producono esportazioni DOT/BIG, il grafo di derivazione (*lineage*), marker temporali, metriche e report utili per analisi e notebook.
- Helper di composizione supportano script e workflow modulari senza spostare la sorgente di verità fuori dal log eventi; per i workflow scriptati, il contratto canonico resta la DSL `session.jsonl`.

Infrastructure (I/O, strumenti esterni, backend)

- L'event store JSONL realizza la persistenza append-only della storia dei trial.

- Un backend fake supporta smoke test, demo e notebook leggeri senza dipendere da hardware o training reale.
- Nel prototipo attuale il training reale su TorchSharp é orchestrato dal modulo `Infrastructure.TorchSharpBackend`, mentre i moduli di proiezione e compilazione verso la rappresentazione eseguibile vivono in `Backend.TorchSharp`; la separazione preserva il confine tra adattatore infrastrutturale ed esecuzione numerica.[25]
- Adattatori dedicati gestiscono i passaggi verso strumenti esterni come BigraphER, mantenendo esplicita l'interfaccia verso l'esterno (*boundary*).[21]

8.3 Pipeline applicativa concreta

La pipeline applicativa reale del prototipo é quella implementata dal command handler:

`load` → `fold evolve` → `decide` → `append`.

Il flusso é semplice, ma é proprio questa semplicità a rendere leggibile il rapporto tra semantica ed esecuzione.

1. **Load.** Per un dato trial, l'Application legge lo stream JSONL corrispondente dall'event store.
2. **Fold / replay.** La storia viene riprodotta tramite `V1Evolve`; il risultato é lo stato corrente del trial prima del nuovo comando.
3. **Decide.** Il comando viene passato alla funzione pura di decisione, che o fallisce con un errore di dominio, oppure produce una lista di eventi canonici.
4. **Append.** Gli eventi prodotti vengono persistiti nello stream corretto; se il comando genera una derivazione, il routing scrive gli eventi nel child stream e aggiunge il marker necessario al parent.

Questa pipeline vale sia per i comandi riferiti al singolo trial sia, indirettamente, per la facciata a livello di esperimento: il comando a livello di esperimento viene prima instradato verso lo stream del trial corretto, poi elaborato con la stessa pipeline, e infine l'esperimento viene ricostruito come proiezione dagli stream aggiornati.

Listing 8.1: Schema semplificato della pipeline applicativa `load -> fold -> decide -> append`.

```
let handleCommand trialId cmd =
    let history =
        EventStore.load trialId

    let currentState =
        history
        |> List.fold V1Evolve.evolve None

```

```
    let newEvents =
        V1Decide.decide currentState cmd

    match newEvents with
    | Error err ->
        Error err
    | Ok events ->
        EventStore.appendMany trialId events
        Ok events
```

Il blocco di codice 8.1 mostra in forma semplificata la pipeline applicativa fondamentale del prototipo: caricamento dello stream, replay dello stato corrente, decisione pura sul comando e persistenza dei nuovi eventi.

Due modalità di replay. Il codice distingue una modalità *lenient*, che preserva il comportamento legacy per stream storici non perfettamente allineati, e una modalità *strict*, che fallisce in presenza di storia invalida prima dell'esecuzione di nuovi comandi. Per la tesi, la modalità stretta é quella più vicina al contratto desiderato di replay verificabile; la modalità permissiva resta soprattutto una scelta di compatibilità del prototipo corrente.

8.4 Persistenza eventi e replay

Nel prototipo corrente il log eventi é serializzato in formato JSONL con *uno stream per trial*. Ogni riga contiene un envelope minimale con campi di primo livello come `ts`, `type`, `cid`, `actor` e `data`; il payload effettivo dell'evento resta determinato dalla coppia `type + data`, mentre gli altri campi servono soprattutto a provenienza, audit e correlazione operativa.[18]

```
{ "ts":"2026-03-05T22:09:02.5398050+00:00",
  "type":"TrialCreated",
  "cid":"10a7b06e08fe4453a523d6dbfa9aa1d7",
  "actor":{"kind":"Human","id":"cli"},
  "data":{
    "trialId":"cmp_t0",
    "dataset":{"name":"demo","version":"v1","split":null,"seed":7},
    "seed":7
  }
}
```

La deserializzazione ricostruisce la DU degli eventi di dominio, che viene poi consumata dal replay. In questo senso il file JSONL non é solo un log di audit: é davvero la sorgente operativa da cui il trial state viene ricostruito.

Comandi ed eventi reali del prototipo. Nel codice del prototipo compaiono, tra gli altri, comandi per creazione, derivazione di trial (ancora esposta in parte anche come `fork` nella superficie pubblica), applicazione di patch, registrazione e applicazione

Famiglia di comando	Eventi principali emessi	Osservazione implementativa
Create trial	`TrialCreated`	Inizializza il trial e crea lo stato di base da cui parte il replay successivo.
Derive / fork	`TrialForked`, `ForkSnapshot`, `ForkedOut`	La derivazione materializza un trial figlio con parentage osservabile; il parent conserva un marker di uscita.
Apply patch	`PatchApplied`	Rappresenta il canale canonico di mutazione del trial; nel caso architetturale aggiorna la componente A.
Register rule	`ReactionRuleRegistered`	Estende il registry locale delle reaction rules senza mutare direttamente il modello.
Apply reaction	`ReactionApplied`, `PatchApplied` (eventualmente dopo `TrialForked`, `ForkSnapshot`)	La reaction registra la provenienza della trasformazione; il cambiamento effettivo passa comunque da `PatchApplied`.
Request train	`BoundaryRequested`, poi `PreTrainMetricsRecorded` / `MetricsRecorded`, `CheckpointRegistered`, `TrainingCompleted`, `BoundaryCompleted`	Il comando apre il *boundary*; metriche, checkpoint ed eventi di completamento vengono prodotti dal backend e riappendati nello stream del trial.

Tabella 8.2: Sintesi delle principali famiglie di comando del prototipo e dei corrispondenti eventi osservabili nello stream del trial.

di reaction rules e richiesta di training; gli eventi concreti includono creazione del trial, derivazione con `TrialForked`/`ForkSnapshot`/`ForkedOut`, patch applicate, richieste e completamenti al *boundary*, metriche pre/post training, checkpoint e completamento del training. Per il lettore della tesi il punto architetturalmente più importante é questo: nel prototipo `PatchApplied` resta l'evento che realizza il cambiamento strutturale effettivo dell'architettura, mentre eventi come `ReactionApplied` fungono soprattutto da provenienza e tracciabilità della decisione trasformativa.

Listing 8.2: Esempio semplificato della discriminated union dei comandi del prototipo.

```
type Command =
    | CreateTrial of TrialId * DatasetRef * int
    | ForkTrial of ParentId * ChildId
    | ApplyPatch of TrialId * PatchMode * Patch
    | RegisterRule of TrialId * RuleId * RuleText
    | ApplyReaction of TrialId * RuleId * MatchId
    | RequestTrain of TrialId * int
```

La Tabella 8.2 riassume, in forma sintetica, le principali famiglie di comando del prototipo e gli eventi canonici che esse producono nel flusso applicativo corrente.

Interfaccia verso l'esterno e training. Le operazioni non pure non vengono eseguite nel Domain. Nel caso del training, il comando produce una richiesta verso l'interfaccia verso l'esterno (*boundary*); l'esecuzione vera avviene nel backend e il suo esito viene riportato nello stream tramite eventi di metrica, checkpoint e completamento. In questo modo il replay non riesegue il training, ma ne ricostruisce gli effetti osservabili sul trial.

Nel caso specifico delle derive con policy `smooth`, l'implementazione corrente mantiene un perimetro volutamente ristretto: la semantica concreta è supportata quando la derivazione materializza uno `SmoothBlock`. In quel caso la continuità col parent non richiede un nuovo formato di checkpoint dedicato: il trasferimento parametrico riusa il medesimo adattamento nominale già impiegato per `preserve`, mentre la transizione graduale è realizzata nel modello compilato tramite rami `Old`/`New` e una schedule di α aggiornata dal backend rispetto al punto di derivazione.

8.5 CLI e superfici operative

La CLI fornisce un punto di ingresso unico per: (i) creare e derivare trial, (ii) applicare patch e reaction rules, (iii) richiedere training su backend fake o TorchSharp, (iv) rigenerare viste e dati per analisi, (v) esportare l'architettura in DOT o in formato `.big`, o richiedere altri artefatti (ad esempio `.csv`).

```
# Creazione e derivazione dei trial
trial create --id <id> --dataset demo --seed 1 ...
derive bare  --parent <id> [--child <child>] ...

# Mutazioni e reaction rules
patch apply    --trial <id> --file <patch.json> ...
rules register --trial <id> --ruleId <rid> --file <rule.txt> ...
react preview  --trial <id> --ruleId <rid> ...
react apply    --trial <id> --ruleId <rid> --matchId <mid> ...

# Training e viste
train request --trial <id> --epochs 5 --backend TorchSharp ...
view diff-arch --trial <id> ...
view report    --trial <id> ...
view plot-data --trial <id> --outdir <dir> ...

# Export architettura
export dot --trial <id> --out <file.dot> ...
export big --trial <id> --out <file.big> ...
```

Nel codice corrente resta disponibile anche `trial fork` come superficie pubblica; tuttavia, per rappresentare la semantica attuale della derivazione, la famiglia `derive`

(`bare`, `patch`, `reaction`) è più fedele allo stato del framework. Notebook, script e workflow JSONL modulari restano superfici operative importanti, soprattutto per i casi d'uso e per la raccolta di evidenze. Tuttavia, nel prototipo attuale la gerarchia va letta in modo esplicito: per i workflow scriptati il contratto ufficiale resta la DSL `session.jsonl`, eseguita dal relativo runner applicativo; la CLI rimane invece la superficie pubblica più stabile per il ciclo di vita di base dei trial, le viste e gli export.

8.6 Determinismo e scelte implementative di robustezza

Il determinismo é essenziale per rendere riproducibili traiettorie, viste ed evidenze. Nel prototipo attuale si adottano soprattutto le seguenti linee guida:

- **Core puro.** `decide` e `evolve` non eseguono I/O e non dipendono da tempo reale o random implicito.
- **Replay esplicito.** Lo stato non viene letto da snapshot opachi del dominio, ma ricostruito dalla storia degli eventi tramite replay.
- **Ordinamenti stabili.** Esportazioni e viste cercano ordinamenti deterministici, per ridurre differenze spurie nei diff e nei report.
- **Seed espliciti.** Le componenti stocastiche del training e dei controller sono guidate da seed espliciti o registrati in configurazione/artefatti.
- **Boundary osservabile.** Gli effetti esterni non alterano direttamente il core: lasciano tracce osservabili nello stream tramite eventi di richiesta, metriche, checkpoint e completamento.

Dove opportuno, il prototipo conserva anche metadati tecnici di correlazione e provenienza; tuttavia, la tesi non dipende da una particolare forma serializzativa di tali metadati, ma dal fatto che replay e audit restino possibili.

8.7 Testing e verifica

La correttezza dell'implementazione é sostenuta soprattutto da una suite xUnit di test di regressione, di caratterizzazione, smoke test e integrazioni mirate sui contratti principali del prototipo. Il progetto include anche il supporto ecosistemico a test generativi in stile QuickCheck tramite FsCheck, ma nel codice attuale il nucleo verificabile é dato principalmente da test esempio/regressione più che da una batteria estesa di property-based tests [4], [9].

- **Replay tests.** Verificano roundtrip dell'event store, ricostruzione coerente dello stato, comportamento strict del replay e coerenza tra replay e viste derivate.
- **Invarianti strutturali.** Verificano validazione di bigrafo e patch, inclusi casi come endpoint dangling o rimozioni strutturalmente non ammissibili.

- **Derivazione e grafo di derivazione.** Verificano che trial derivati, parentage e marker osservabili restino coerenti con la storia degli eventi prodotta.
- **Boundary / backend.** Smoke test e test di integrazione mirati controllano il flusso training request → metriche/checkpoint/completamento, inclusi casi di resume sul backend TorchSharp e casi leggeri su backend fake.

Il perimetro della verifica resta quindi concreto ma mirato: sostiene i contratti implementativi principali del prototipo, senza equivalere a una copertura esaustiva di tutti i workflow end-to-end.

Capitolo 9

Valutazione property-driven

La valutazione di questa tesi è *property-driven*: l'obiettivo non è dimostrare prestazioni SOTA su benchmark, ma verificare proprietà del framework che rendono la ricerca AutoML *tracciabile* e *verificabile*. Le proprietà (oracoli) derivano dai requisiti (Capitolo 4), dalla semantica a eventi (Capitolo 5) e dai casi d'uso UC1–UC3 (Capitolo 7). Il focus è quindi su *correttezza del replay*, *confronti rigenerabili rispetto a una vista*, *coerenza dell'interfaccia verso l'esterno* (*boundary*), *intercambiabilità dei controller* e *verificabilità della derivazione* in presenza di strumenti e backend esterni. [8], [16], [18] Questa impostazione è già esercitata da un insieme di tracce dimostrative leggere, che coprono replay, confronto, derivazione e metadati di controller senza dipendere da benchmark prestazionali pesanti. Quando servono invece istanziazioni sperimentali più ricche, con artefatti sperimentali discussi in forma narrativa, il riferimento naturale è Capitolo 10.

9.1 Domande di valutazione

Le domande principali sono:

- **E1 — Correttezza del replay.** Il replay ricostruisce lo stesso stato del trial e le stesse proiezioni rigenerabili a partire dallo stesso log (più artefatti referenziati)?
- **E2 — Correttezza del confronto rispetto a una vista.** I confronti rigenerati rispetto a viste scelte (Δ_V) sono coerenti con i cambiamenti osservabili tra stati del trial e tra modelli confrontati?
- **E3 — Coerenza del boundary.** Il roundtrip verso strumenti esterni è *verificabile* e compatibile con il replay (input/complement/output identificati, controlli di rientro registrati, import deterministico)?
- **E4 — Intercambiabilità dei controller.** Cambiare la sorgente dei comandi (controller manuale, `Human`, oppure controller automatico, `PolicyAgent`) modifica solo la provenienza (`meta`), non la semantica del dominio e non gli stati ricostruiti.

- **E5 — Verificabilità della derivazione.** Un trial derivato può essere ricondotto in modo verificabile al punto di derivazione dichiarato e alla baseline comune da cui deriva?

Nota su prestazioni e overhead. La misurazione accurata dell'overhead non è un obiettivo primario: questa tesi presenta un prototipo attuale, non ottimizzato, e numeri assoluti sarebbero potenzialmente fuorvianti. L'overhead viene quindi discusso qualitativamente, indicando i fattori dominanti e le possibili ottimizzazioni. [20]

9.2 Setup sperimentale minimo e riproducibile

La valutazione usa un setup volutamente semplice, sufficiente a esercitare le proprietà chiave. L'enfasi è sulla riproducibilità: ogni scenario produce un log eventi e un insieme di artefatti, e può essere ripetuto via replay deterministico.

- **Backend.** Prototipo attuale su TorchSharp; per gli oracoli minimi discussi qui si usano anche tracce con backend `fake`, utili per isolare la semantica del framework e ottenere log deterministici di training, checkpoint e completamento del boundary. [25]
- **Task/Dataset.** Il task minimo usato per la validazione semantica è volutamente leggero: nelle tracce demo il dataset è registrato come `demo` e produce metriche elementari di `acc`/`loss`, sufficienti per esercitare replay, checkpoint ref, derivazione e confronti senza spostare il focus su performance assolute.
- **Scenari.** UC1–UC3 come scenari principali: (i) riscrittura interna, (ii) round-trip di boundary, (iii) derivazione e confronto tra trial. Le tracce dimostrative oggi disponibili coprono già la parte minimale di questi scenari sul piano degli oracoli.

Artefatti prodotti. Ogni esecuzione produce: (i) un log JSONL con metadati e payload di dominio, (ii) checkpoint e metriche come artefatti referenziati, (iii) report di confronto rispetto a una vista (proiezioni), (iv) export/import per il boundary e (v) visualizzazioni (DOT e rappresentazioni `.big`). Questi artefatti sono gli input degli oracoli e vengono tracciati tramite riferimenti e, quando disponibili, impronte nel log. Nel contratto degli artefatti, il log resta la fonte primaria, mentre report, confronti, viste e visualizzazioni sono superfici derivate e rigenerabili. In questo capitolo tali superfici contano soprattutto come evidenze per gli oracoli; le loro istanziazioni sperimentali più complete vengono invece riprese in Capitolo 10.

9.3 Metodologia e oracoli di verifica

In questa sezione le proprietà sono presentate come oracoli deterministici che consumano *solo* log e artefatti persistiti. L'approccio è affine al testing basato su proprietà: per ciascun requisito si definisce una proprietà verificabile, con controesempi espressi come tracce di eventi. [4], [9]

Oracolo	Proprietà verificata	Input principale	Esito	Artefatto / evidenza
Replay correctness	Il replay del log ricostruisce uno stato coerente con la storia del trial	Log eventi, riferimenti agli artefatti	PASS	Report di replay, stato ricostruito
Compare-under-view correctness	Il confronto sotto una vista è coerente con il cambiamento osservato	Due stati del trial, vista selezionata	PASS	Report di confronto, export tabellare
Boundary coherence	Il roundtrip esterno produce un rientro verificabile e coerente	Artefatto esportato, risultato esterno, stato reintegrato	PASS	Validation report, artefatti di boundary
Controller interchangeability	Cambiare controller non altera il significato del dominio a parità di azione	Due tracce con meta diverse e stesso payload di dominio	PASS	Confronto tra replay, report comparativo
Lineage coherence	Parentage, baseline comune e derivazioni restano coerenti	Log del trial, riferimenti a parent e child	PASS	Lineage report, grafo dei trial

Tabella 9.1: Oracoli concettuali della valutazione. Ogni oracolo rende osservabile una proprietà del framework tramite un controllo deterministico su log, stati derivati e artefatti persistiti.

La Tabella 9.1 sintetizza gli oracoli concettuali usati nella valutazione e il tipo di evidenza osservabile associata a ciascuna proprietà. Mentre la Tabella 9.2 collega ciascuna domanda E1–E5 agli oracoli attivati e agli output osservabili prodotti nelle tracce, rendendo esplicito quali controlli vengono esercitati in ogni scenario.

9.3.1 Correttezza del replay (E1)

Metodo. Dato un log eventi $E^{(t)}$ di un trial t, si esegue replay completo ricostruendo lo stato finale $T_n^{(t)}$. Si confrontano quindi:

- impronta dell'architettura $\mathrm{fp}(\pi_{V_A}(A))$ (serializzazione canonica della vista architetturale),
- snapshot deterministico degli iperparametri $\pi_{V_H}(H)$,
- integrità dei riferimenti: W (checkpoint ref) e artefatti osservazionali in O (metriche/export/import).

Oracolo. A parità di log e artefatti referenziati, il replay deve produrre lo stesso stato del trial (impronte e riferimenti) e le stesse proiezioni rigenerabili. Differenze

Scenario	Obiettivo	Oracoli usati	Output osservabili
E1	Verificare la correttezza del replay del trial	Replay correctness	Replay report, stato ricostruito, confronto con gli artefatti attesi
E2	Verificare la coerenza del confronto sotto una vista selezionata	Compare-under-view correctness	Report di confronto, esportazione tabellare, evidenza del cambiamento osservato
E3	Verificare la correttezza del roundtrip verso uno strumento esterno	Boundary coherence	Validation report, artefatti di export/import, stato reintegrato
E4	Verificare l'intercambiabilità del controller a parità di payload di dominio	Controller interchangeability	Confronto tra replay, report comparativo, evidenza di invarianza del dominio
E5	Verificare la coerenza del lineage tra baseline comune e trial derivati	Lineage coherence	Lineage report, grafo dei trial, evidenza di parentage e derivazione

Tabella 9.2: Scenari di valutazione e relativi oracoli. Ogni scenario attiva uno o più controlli osservabili, i cui esiti vengono materializzati in report, esportazioni o altri artefatti leggibili.

sono ammissibili solo se esplicitate nel log (es. migrazioni di `schemaVersion` o cambi di strumento/versione registrati nella provenienza).

9.3.2 Correttezza del confronto rispetto a una vista (E2)

Metodo (confronto locale su trace). Per ogni passo consecutivo del trial $T_k^{(t)} \to T_{k+1}^{(t)}$, si rigenera un confronto locale rispetto a una vista scelta:

$$\Delta^{\text{step}}_{V,t,k\to k+1} := \Delta_V\Big(T_k^{(t)}, T_{k+1}^{(t)}\Big),$$

dove la vista V viene applicata al trial ricostruito. Quando il confronto usa la vista di modello V_M, il modello entra quindi come proiezione derivata del trial, non come oggetto primario della semantica. Per viste che includono W (es. V_M o V_{core} quando il riferimento ai pesi è osservabile), il confronto è inteso sui *riferimenti* (checkpoint ref/impronta), non sui tensori.

Metodo (confronto comparativo tra trial). Si considerano inoltre confronti non consecutivi (anche tra trial diversi), ad esempio baseline vs finale, o trial A vs trial B:

$$\Delta_V(T_a, T_b).$$

Questo abilita confronti mirati: sola architettura (V_A), soli iperparametri (V_H), nucleo di progetto (V_{core}) e vista di modello $V_M(T) = M(T) = \langle A, W \rangle$ quando serve confrontare candidati allenati.

Oracolo. L'etichetta tecnica può restare *diff soundness* nelle superfici implementative o nei report derivati, ma nel testo la leggiamo come correttezza del confronto rispetto a una vista. Il confronto persistito (se presente come read model) deve essere rigenerabile e coerente con i confronti calcolati sul replay. Inoltre, per classi di eventi "di dominio" (riscritture interne, set iperparametri, import di boundary) il confronto deve riflettere cambiamenti osservabili in modo consistente con la vista scelta.

9.3.3 Coerenza del boundary (E3)

Metodo. Si eseguono roundtrip di boundary (UC2) e si controlla:

- presenza e correttezza di `toolId/toolVersion` e input dichiarati nella provenienza,
- impronte coerenti per input, complement e output,
- esito dei controlli di rientro (invarianti) registrato,
- import compatibile con replay: il replay ricostruisce lo stesso stato importato senza rieseguire lo strumento esterno.

Oracolo. Ogni roundtrip deve produrre un ΔBoundary completo e verificabile; eventuali violazioni di invarianti devono risultare esplicite (errori o eventi di rifiuto), così da evitare stati "mezzi importati". [7]

9.3.4 Intercambiabilità dei controller (E4)

Metodo. Nel prototipo attuale l'evidenza curata usa uno scenario host di mutazione interna e costruisce una controparte sintetica del workflow sostituendo la provenienza dei soli eventi emessi dal controller manuale `Human` con un controller automatico `PolicyAgent` stub, lasciando invariati `type + data`. In questo modo non si introduce un secondo caso pesante: si verifica la proprietà E4 sullo stesso scenario già usato per UC1, replay e confronto sotto vista.

Il confronto avviene quindi tra lo stream originale e il suo mirror di controller su:

- payload di dominio degli eventi (uguali, a parità di sequenza),
- stato del trial ricostruito via replay (uguale),
- viste rigenerate dal replay (data-view ed executable-view) a partire dai due stream,
- metadati `meta` (diversi per provenienza: `Human` in modalità manuale vs `PolicyAgent` automatico).

Oracolo. Il cambiamento di controller deve riflettersi solo nella provenienza; il replay e le proiezioni devono essere invarianti rispetto ai metadati. In altre parole: *la semantica del dominio dipende solo dal payload*, mentre i metadati servono per audit, attribuzione di provenienza e tracciabilità. [28]

9.3.5 Verificabilità della derivazione (E5)

Metodo. Nel contratto attuale del prototipo, per ogni trial derivato si recuperano dal log almeno `childTrialId`, `parentTrialId`, `atEpoch` e l'eventuale riferimento al checkpoint di partenza. Si controlla quindi che il child stream inizi con un evento tecnico di derivazione come `TrialForked` coerente con il parent dichiarato e che il parent stream riporti il corrispondente `ForkedOut` allo stesso `atEpoch`. Quando la derivazione nasce da una baseline comune usata per confrontare due rami, si verifica inoltre che i confronti tra trial siano ancorati allo stesso punto di derivazione dichiarato.

In questa sede basta richiamare che la derivazione può dipendere anche dalla weight policy associata al child, e che `smooth` è trattata nel prototipo come forma controllata di transizione graduale. Il significato concettuale di queste policy appartiene al quadro semantico della derivazione (Capitolo 5), mentre le loro evidenze sperimentali si leggono meglio nei case study di Capitolo 10.

Oracolo. La derivazione è valida se la trace rende coerente e controllabile il punto di derivazione dichiarato su entrambi i lati del *lineage*, ossia del grafo di derivazione: child e parent devono concordare su parent id e `atEpoch`, e i confronti tra rami devono partire dalla stessa baseline dichiarata. In questo modo il *lineage* non resta un'informazione narrativa, ma diventa una proprietà controllabile del framework. Una certificazione più forte mediante hash dello stato base resta un'estensione naturale, ma non viene assunta come prerequisito del prototipo.

Listing 9.1: Esempio semplificato di lineage oracle in F#.

```
type OracleResult =
    { Name: string
      Ok: bool
      Messages: string list }

let private pass name msgs =
    { Name = name; Ok = true; Messages = msgs }

let private fail name msgs =
    { Name = name; Ok = false; Messages = msgs }

let runLineageOracle (log: TrialLog) : OracleResult =
    let st = log |> replayStrict
    let errors =
        [ if st.Lineage.Id <> st.TrialId then
              yield "TrialId e lineage.Id non coincidono."
          if st.Lineage.Parent = Some st.TrialId then
```

```
18                yield "Un trial non puo' avere se stesso come
                      parent."
19            match st.Lineage.Source, st.Lineage.Parent with
20            | "create", Some _ -> yield "Un trial creato non
                  deve avere parent."
21            | "derive", None   -> yield "Un trial derivato deve
                  avere un parent."
22            | _ -> () ]
23    if List.isEmpty errors then
24        pass "Lineage coherence" [ "Lineage coerente." ]
25    else
26        fail "Lineage coherence" errors
```

9.4 Conclusioni sui risultati di valutazione

Questa sezione raccoglie i risultati per E1–E5 in forma sintetica, collegandoli ai casi d'uso UC1–UC3 e agli artefatti prodotti (log, report di confronto, export/import). Per ciascuna domanda: (i) si presenta un output minimo (tabella/figura), (ii) si collega l'evidenza al requisito corrispondente, (iii) si annotano limiti e modalità di fallimento. Le evidenze qui richiamate restano volutamente compatte e orientate agli oracoli; quando il focus si sposta su artefatti sperimentali completi e sulla lettura congiunta di timeline, metriche, lineage e viste strutturali, il rimando naturale è a Capitolo 10.

La Tabella 9.3 riassume le cinque proprietà E1–E5, lo scenario host più leggibile per ciascuna e l'artefatto osservabile usato come evidenza primaria.

9.4.1 E1: correttezza del replay

Le tracce dimostrative oggi disponibili mostrano già il caso minimo richiesto: a partire da un log con `TrialCreated`/eventi tecnici di derivazione, metriche, checkpoint e completamenti di boundary, il replay può ricostruire stato, epoca logica e riferimenti agli artefatti senza rieseguire il backend. Questa catena è esercitata anche da scenari di mutazione interna in cui stream eventi, esportazioni strutturali e report comparativi restano tutti rigenerabili a partire dalla stessa storia. Gli scenari di roundtrip di boundary aggiungono inoltre la variante in cui il replay deve mantenere stabili anche i riferimenti agli artefatti di attraversamento del boundary, senza rieseguire lo strumento esterno.

9.4.2 E2: correttezza del confronto rispetto a una vista

Un esempio minimo di confronto già disponibile è costituito da scenari che isolano una sola mutazione architetturale su un child derivato e materializzano il confronto osservabile in tre forme complementari: esportazioni prima/dopo sotto vista architetturale, tabelle comparative di metriche e marker, e sequenza dei passi che delimita il punto di cambiamento. In questo senso i report persistiti non vanno letti come semplici dump tecnici, ma come istanze concrete di Δ_V: la vista architetturale

Proprietà	Scenario host	Artefatto osservabile
E1 — correttezza del replay	scenario di mutazione interna, con rinforzo da roundtrip di boundary	stream eventi persistiti, esportazioni strutturali e riferimenti agli artefatti di boundary rigenerabili via replay
E2 — confronto rispetto a una vista	scenario di mutazione interna a baseline comune	esportazioni architetturali prima/dopo, tabelle comparative di metriche e sequenze selettive dei passi come istanze concrete di Δ_V
E3 — coerenza del boundary	scenario di roundtrip di boundary	sintesi del roundtrip, report di validazione persistito, export/import eseguibili e conferma del rientro verificabile
E4 — intercambiabilità dei controller	scenario di mutazione interna con mirror di controller	report di equivalenza controller e viste rigenerate identiche dopo il mirror sintetico della provenienza
E5 — verificabilità della derivazione	scenario di derivazione controllata a baseline comune	lineage, sequenze selettive dei passi, metriche finali e report che ancorano i rami alla stessa baseline dichiarata

Tabella 9.3: Sintesi finale delle proprietà valutate, dello scenario host oggi più leggibile e dell'artefatto osservabile usato come evidenza primaria.

rende leggibile la differenza strutturale, mentre le viste metriche mostrano l'effetto osservabile della stessa mutazione sul comportamento del trial.

9.4.3 E3: coerenza del boundary

Le tracce demo contengono già la forma minima del contratto di boundary per il training: `BoundaryRequested` registra input e piano di esecuzione, `BoundaryCompleted` registra output e stato finale, e il replay ricostruisce gli stessi riferimenti senza rieseguire il backend. A questo si aggiungono scenari curati di roundtrip che forniscono una evidenza UC2 più leggibile: export di una vista eseguibile, import con proiezione verificabile, controllo di rientro e report di validazione persistito.

Questa evidenza va letta come prova della forma *lens-inspired* oggi realmente implementata: input e output del boundary sono tracciati e verificabili, mentre un complement first-class e una reintegrazione pienamente bidirezionale nel senso forte di *lens with complements* restano ancora fuori dal perimetro consolidato del framework. Nelle tracce curate oggi disponibili, la verifica minima osservabile è già forte: il report di validazione conferma roundtrip byte-identical della vista eseguibile, stessa proiezione dei layer e presenza sia di `BoundaryRequested` sia di `BoundaryCompleted` nel log del trial sorgente. Una istanziazione sperimentale più ricca di questo stesso punto è rimandata a Capitolo 10.

L'evidenza minima leggibile per questo caso è già disponibile come sintesi del roundtrip, report di validazione persistito e tabella riassuntiva dei controlli di rientro; una figura finale può essere derivata direttamente da questi stessi artefatti senza cambiare il contratto del caso sperimentale.

9.4.4 E4: intercambiabilità dei controller

Le tracce già mostravano la separazione richiesta: eventi come `TrialCreated`, eventi tecnici di derivazione o `BoundaryRequested` possono essere emessi da un controller manuale, mentre metriche, checkpoint e completamenti di training provengono da controller o backend automatici. Questa proprietà è ora chiusa da un piccolo artefatto dedicato, ospitato dallo stesso scenario di mutazione interna: il report di equivalenza costruisce una copia speculare degli stream del caso sostituendo la provenienza dei soli eventi emessi da `Human` con un `PolicyAgent` stub, lasciando invariati `type + data`. La tabella riassuntiva risultante mostra che i payload restano identici, le viste rigenerate dal replay hanno la stessa impronta e l'ultimo passo metrico coincide per entrambi i trial del caso. In questo modo la differenza resta confinata ai metadati di provenienza, come richiesto da E4.

9.4.5 E5: verificabilità della derivazione

Le tracce di derivazione e confronto oggi disponibili mostrano già la forma minima di questa proprietà: il child stream dichiara il parent e l'epoca di derivazione tramite un evento tecnico come `TrialForked`, mentre il parent stream registra il corrispondente `ForkedOut`. Questo basta per rendere verificabile il punto di derivazione e per ancorare i confronti tra rami a una baseline comune dichiarata, pur senza introdurre ancora una certificazione hash-based dello stato base.

Discussione qualitativa su overhead. L'overhead dipende principalmente da: (i) dimensione del log (numero eventi), (ii) dimensione degli artefatti (checkpoint/metriche), (iii) costo delle proiezioni e dei report di confronto e (iv) costo dei roundtrip di boundary. Nel prototipo, il costo dominante resta il training; la serializzazione del log e la generazione di report sono proporzionali al numero di eventi e alle dimensioni delle viste. Una quantificazione empirica più precisa può essere aggiunta in chiusura, ma il punto centrale per questa tesi resta che i costi introdotti dalla tracciabilità sono lineari negli eventi e dominati, negli scenari attuali, dal costo del training e dei roundtrip esterni.

9.5 Minacce alla validità

- **Generalizzazione.** La valutazione usa task/dataset minimi; i risultati sulle proprietà sono indipendenti dal task, ma prestazioni assolute non sono l'obiettivo.
- **Evoluzione di strumenti e backend.** Strumenti esterni e librerie possono evolvere; per questo si registrano versione e impronte e si trattano come parte della provenienza.
- **Non-determinismo del training.** Alcuni backend possono introdurre non determinismo; la tesi si concentra sul determinismo del core e sul tracciamento esplicito dei fattori esterni (seed, versioni, piattaforma).

- **Copertura degli oracoli.** Gli oracoli verificano proprietà strutturali e contrattuali; non coprono esaustivamente tutte le modalità di guasto possibili in sistemi ML complessi.

Capitolo 10

Risultati: casi di studio sperimentali e artefatti prodotti dal framework

Questo capitolo esiste per affiancare alla valutazione guidata da proprietà del capitolo precedente una lettura più concreta degli artefatti che il framework produce quando viene usato su esperimenti reali. Se in Capitolo 9 l'obiettivo era verificare proprietà del sistema tramite oracoli e scenari minimi, qui l'obiettivo è diverso: mostrare in forma leggibile quali esperimenti maturi il framework riesce già a generare, conservare e rendere riusabili dentro la narrazione tecnica della tesi. Il capitolo non introduce quindi nuovi oracoli e non anticipa ancora la discussione di limiti o sviluppi futuri, che resta materia di Capitolo 11.

Il criterio editoriale è volutamente selettivo. Per ogni esperimento mostriamo solo gli artefatti che aiutano davvero a leggere la storia del trial, inteso anche qui come unità sperimentale primaria del framework: una selezione di eventi chiave dell'archivio di eventi, una vista sintetica delle metriche, una rappresentazione della struttura o del grafo di derivazione, e una tabella finale compatta. I file inclusi in tesi derivano direttamente dagli artefatti originari dell'esperimento oppure da trasformazioni automatiche stabili dei relativi sorgenti DOT e CSV. Qui mostriamo quattro casi di studio, scelti perché complementari e già sostenuti da esperimenti stabili: una derivazione controllata con finalizzazione della mutazione architetturale smooth, un confronto tra derivazione architetturale con transizione smooth e derivazione architetturale con transizione hard verso CNN, una ricerca multi-round in cui selezione, grafo di derivazione e artefatti narrativi restano osservabili dentro lo stesso esperimento, e infine un esperimento puramente strutturale in cui il valore dimostrativo non sta nel training ma nella coerenza tra bigrafo, vista strutturale, grafo di derivazione ed export degli artefatti.

10.1 Contesto sperimentale comune

I casi di studio qui raccolti condividono un contesto sperimentale abbastanza uniforme da renderne leggibile il confronto, pur restando diversi per struttura narrativa e per forma del grafo di derivazione osservato. Nei primi tre casi il task concreto è

FashionMNIST, con training reale su backend TorchSharp e con artefatti persistiti che rendono confrontabili trial, metriche e passaggi di derivazione dentro lo stesso esperimento.

- **Dataset e task.** I primi tre esperimenti lavorano su FashionMNIST: il trial radice parte da una baseline lineare e i trial derivati esplorano trasformazioni architetturali o variazioni iperparametriche sullo stesso task classificativo.
- **Protocollo sperimentale minimo.** Il protocollo condiviso resta leggero e riproducibile: ogni esperimento dichiara nel proprio `experiment_state.json` il budget di training, i trial coinvolti, il grafo di derivazione e gli artefatti derivati poi inclusi in tesi sotto forma di tabelle, grafici e viste strutturali.
- **Metriche e vincoli.** Le metriche osservate in modo coerente nei diversi casi eseguibili sono soprattutto `train_loss`, `val_loss`, `train_acc` e `val_acc`; i budget sono intenzionalmente contenuti e dichiarati per esperimento, in modo da privilegiare leggibilità della traiettoria, costo sperimentale moderato e rigenerabilità degli artefatti rispetto a benchmark prestazionali estesi.
- **Criterio di scelta dei casi.** I casi inclusi sono complementari: il primo mostra una derivazione controllata con finalizzazione della mutazione architetturale smooth; il secondo rende leggibile una derivazione con due rami concorrenti, smooth e hard; il terzo espone una ricerca multi-round in cui il trial selezionato di ogni round viene scelto rispetto alla vista osservazionale dell'accuratezza di validazione; il quarto mostra invece un esperimento puramente strutturale, senza training reale, ma con architetture e riscritture più complesse.

10.2 Caso di studio 1: mutazione architetturale smooth da baseline lineare verso MLP finale

Il primo caso di studio usa l'esperimento `thesis-linear-smooth-mlp`, costruito per rendere leggibile una storia di derivazione in tre passi: un trial radice lineare, un trial figlio ottenuto tramite mutazione architetturale smooth, e un trial finale in cui la mutazione viene finalizzata. Il valore del caso non sta nel benchmark prestazionale assoluto, ma nella capacità di mostrare insieme timeline, metriche, grafo di derivazione e artefatti strutturali all'interno di uno stesso esperimento.

10.2.1 Scopo e configurazione minima

Lo scopo del caso è osservare una derivazione controllata in cui la baseline iniziale continua a essere allenata dopo la generazione del figlio, mentre il trial smooth e il trial finale seguono una traiettoria propria. La configurazione dichiarata nel file `experiment_state.json` resta volutamente leggera e specifica per l'esperimento: fissa budget distinti per baseline, prosecuzione del ramo radice, trial smooth, trial finale e durata della mutazione smooth, senza trasformare questo caso in un benchmark prestazionale. La mutazione architetturale principale porta da un classificatore

No.	Trial	Action	Epoch	Summary
1	Root	Create trial		Trial created
2	Root	Apply mutation	0	Bootstrap canonical linear baseline from MlProfileBuilders
3	Root	Complete training	29	Training phase completed
4	Smooth	Derive trial	29	Derived from Root with preserve policy
5	Root	Complete training	89	Training phase completed
6	Smooth	Apply mutation	29	Smooth mutation: linear -> MLP(hidden=[128,64]) via SmoothBlock(anchor=0,dur=5)
7	Smooth	Complete training	58	Training phase completed
8	Final	Derive trial	59	Derived from Smooth with preserve policy
9	Final	Apply mutation	59	Finalize smooth: collapse all SmoothBlocks to New branch [early finalize: alpha<AlphaEnd; exact continuity not expected]
10	Final	Complete training	87	Training phase completed

Tabella 10.1: Eventi chiave del caso di studio `thesis-linear-smooth-mlp`. La tabella è derivata automaticamente dagli stream eventi dell'esperimento tramite la proiezione dei passi canonici.

lineare a una MLP con hidden layers $[128, 64]$, poi finalizzata in una variante MLP esplicita.

10.2.2 Timeline di eventi e lettura del grafo di derivazione

La timeline chiave è ricavata automaticamente dagli stream eventi dell'esperimento, senza riportare il dump completo dei file JSONL. Si osservano quattro momenti centrali: creazione della baseline, derivazione del trial smooth dal trial radice, applicazione della mutazione smooth, e derivazione/finalizzazione del trial finale. Questa selezione basta a rendere leggibile il passaggio da una baseline comune a due stati successivi fortemente imparentati. La Tabella 10.1 riassume proprio questa sequenza minima e rende osservabile la storia canonica del caso senza appesantire il testo con il dump completo degli stream eventi.

La struttura genealogica è semplice ma informativa: il trial *Root* agisce come baseline comune, *Smooth* nasce come figlio diretto con politica di preservazione dei pesi, e *Final* deriva a sua volta dal trial smooth per collassare i blocchi smooth in una MLP finale. In questo modo il grafo di derivazione non resta una nota narrativa ma diventa un artefatto osservabile, coerente con il modello del trial completo introdotto nei capitoli precedenti.

10.2.3 Metriche principali e artefatti strutturali

Le curve di accuratezza mostrano che la baseline lineare continua a migliorare dopo il fork, ma i trial derivati raggiungono una qualità finale leggermente superiore

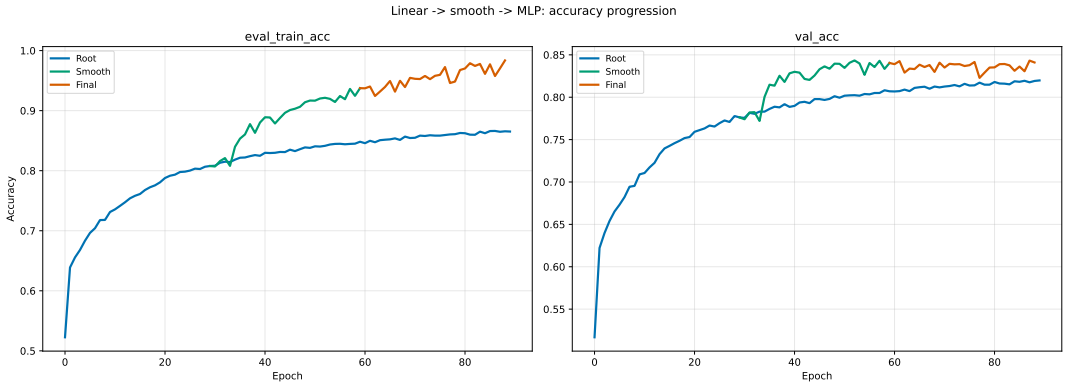


Figura 10.1: Curve di accuratezza del caso di studio `thesis-linear-smooth-mlp`. L'immagine è inclusa direttamente dal grafico prodotto dal flusso di lavoro sperimentale.

Figura 10.2: Grafo di derivazione completo del caso di studio `thesis-linear-smooth-mlp`, generato automaticamente a partire dal sorgente DOT prodotto dal runner sperimentale.

sull'insieme di validazione. La progressione più interessante non è tanto il salto numerico assoluto, quanto il fatto che l'esperimento rende confrontabili tre stati diversi della stessa storia sperimentale: baseline, mutazione smooth in corso e architettura finale stabilizzata. La Figura 10.1 mostra questa progressione sul piano dell'accuratezza, la Figura 10.2 ne esplicita il grafo di derivazione, e la Figura 10.3 completa il confronto sul versante della loss.

La tabella finale conferma questa lettura. Il trial smooth raggiunge la miglior loss di validazione, mentre il trial finale ottiene la miglior accuratezza di validazione tra i tre stati riportati; il trial radice resta comunque importante perché fissa la baseline condivisa da cui la storia prende avvio. L'insieme di questi artefatti mostra bene che il framework non produce solo log o checkpoint isolati, ma esperimenti coerenti in cui metriche, eventi e struttura restano allineati. In particolare, la Figura 10.4 rende visibile la continuità strutturale tra baseline lineare, trial smooth intermedio e MLP finale collassata.

10.2.4 Che cosa dimostra rispetto al framework

Questo primo caso dimostra soprattutto tre aspetti del framework. Primo, la derivazione di trial è leggibile come storia concreta e non solo come proprietà da verificare a posteriori. Secondo, la mutazione smooth genera un percorso sperimentale osservabile con artefatti sia temporali sia strutturali. Terzo, l'esperimento finale è sufficiente

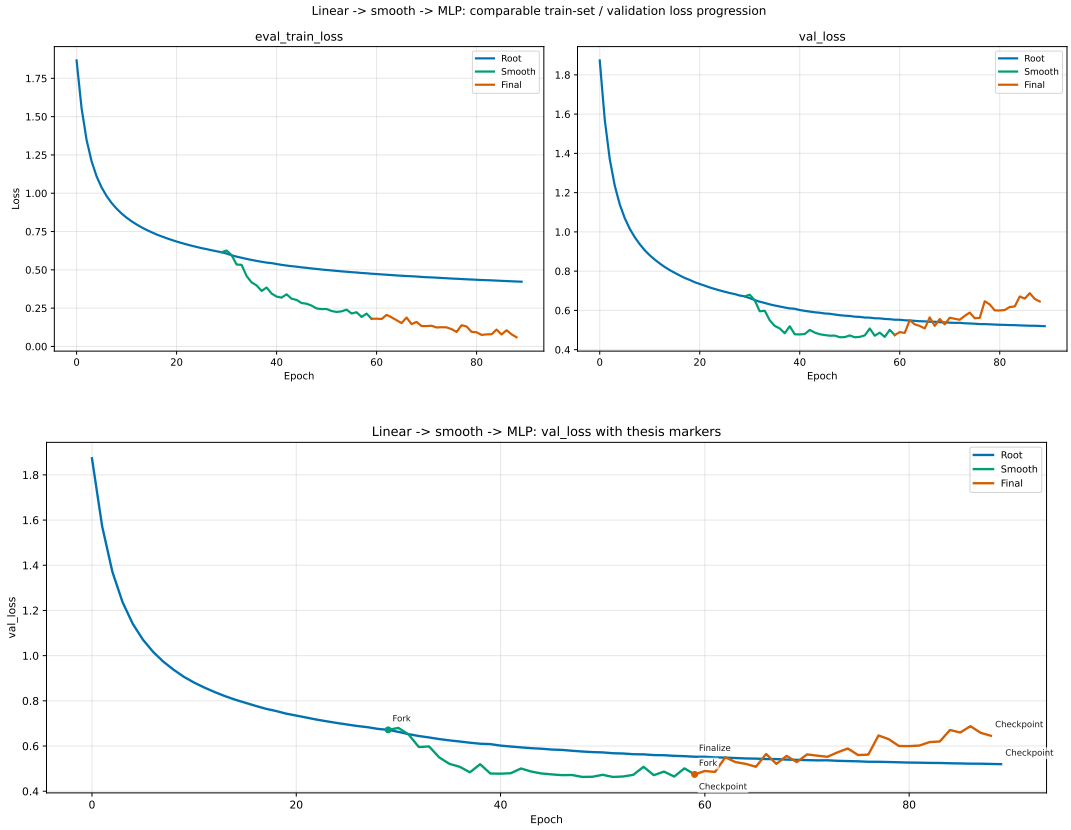
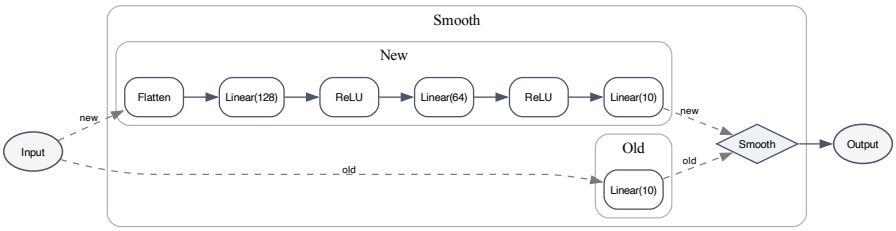


Figura 10.3: Curve di loss e marker di loss di validazione del caso di studio `thesis-linear-smooth-mlp`. La prima vista mostra l'andamento completo di training e validazione; la seconda evidenzia i marker degli eventi principali usati nella tabella riassuntiva.

a sostenere contemporaneamente reportistica leggibile, confronto riproducibile e inclusione diretta nella tesi.

(a) Baseline lineare



(b) Trial smooth con ramo `Old/New`

(c) MLP finale collassata

Figura 10.4: Snapshot strutturali del caso di studio `thesis-linear-smooth-mlp`. Le viste sono impilate verticalmente per rendere esplicita la trasformazione dalla baseline al trial smooth intermedio, fino all'architettura finale stabilizzata. I sorgenti DOT sono esportati dal framework con label di layer compatte e leggibili.

10.3 Caso di studio 2: mutazione architetturale smooth e mutazione architetturale hard verso CNN

Il secondo caso di studio riguarda un esperimento costruito per rendere leggibile una storia di derivazione in cui una baseline lineare genera due sviluppi distinti: da un lato una mutazione architetturale smooth che passa per MLP, finalizzazione e successiva transizione smooth di recupero verso CNN; dall'altro un ramo architetturale hard verso CNN ottenuto con politica di reset dei pesi. Il valore del caso non sta nel benchmark prestazionale assoluto, ma nella possibilità di mettere in relazione dentro lo stesso esperimento: grafo di derivazione, politica dei pesi, artefatti strutturali, traiettorie metriche e marker espliciti delle tre fasi della transizione smooth.

10.3.1 Scopo e configurazione minima

Lo scopo del caso è osservare come il framework renda confrontabili due forme diverse di derivazione architetturale. Il trial radice lineare viene addestrato per poche epoche, poi genera un figlio smooth verso MLP con preservazione dei pesi e un figlio hard verso CNN con reset dei pesi; il trial smooth viene quindi finalizzato in una MLP esplicita e usato come genitore di una seconda mutazione smooth verso una CNN leggera. La configurazione dichiarata nel file `experiment_state.json` resta volutamente leggera e leggibile: `initial_epochs=4`, `smooth_epochs=12`, `final_epochs=8`, `hard_cnn_epochs=40`, con `smooth_anchor_epochs=2` e `smooth_duration=8` per entrambe le transizioni smooth. Questa scelta rende osservabili tre stati distinti della transizione: una fase iniziale ancora in `alpha=0`, l'avvio della rampa smooth e l'ingresso nello stato `alpha=1`.

10.3.2 Timeline di eventi e lettura del grafo di derivazione

La timeline chiave è ricavata automaticamente dagli stream eventi dell'esperimento, senza riportare il dump completo dei file JSONL. I momenti più informativi sono: creazione della baseline, derivazione dei due figli concorrenti dal trial radice, applicazione della mutazione smooth lineare → MLP e della mutazione hard lineare → CNN, derivazione del trial MLP finale con finalizzazione del blocco smooth, e derivazione del trial di recovery smooth MLP → CNN. La tabella seguente mostra che il ramo hard è una vera derivazione architetturale con politica di reset, mentre il ramo smooth mantiene una storia più lunga e articolata, fino alla seconda transizione smooth verso CNN. La Tabella 10.2 rende esplicita questa sequenza minima di eventi e fissa la lettura della storia prima di passare agli artefatti grafici e metrici.

La struttura genealogica è più ricca del classico caso lineare → smooth → final, ma resta leggibile: il trial *Root* agisce come baseline comune, *Smooth MLP* e *Hard CNN* sono due figli fratelli con semantica diversa, *Overfit MLP* finalizza il primo blocco smooth, e *Smooth CNN* deriva infine dall'MLP finale con una seconda mutazione smooth. In questo modo il grafo di derivazione non resta un metadato accessorio, ma diventa un artefatto capace di rendere leggibile anche la differenza tra politiche di derivazione e traiettorie metriche successive.

No.	Trial	Action	Epoch	Summary
1	Root	Create trial		Trial created
2	Root	Apply mutation	0	Bootstrap canonical linear baseline from MlProfileBuilders
3	Root	Complete training	3	Training phase completed
4	Smooth MLP	Derive trial	3	Derived from Root with preserve policy
5	Hard CNN	Derive trial	3	Derived from Root with reset policy
6	Root	Complete training	43	Training phase completed
7	Smooth MLP	Apply mutation	3	Smooth mutation: linear -> MLP(hidden=[64]) via SmoothBlock(anchor=2,dur=8)
8	Hard CNN	Apply mutation	3	Canonical mutation: linear -> lightweight CNN baseline (in=784, out=10)
9	Smooth MLP	Complete training	14	Training phase completed
10	Hard CNN	Complete training	41	Training phase completed
11	Overfit MLP	Derive trial	15	Derived from Smooth MLP with preserve policy
12	Overfit MLP	Apply mutation	15	Finalize smooth: collapse all SmoothBlocks to New branch [early finalize: alpha<AlphaEnd; exact continuity not expected]
13	Overfit MLP	Complete training	21	Training phase completed
14	Smooth CNN	Derive trial	22	Derived from Overfit MLP with preserve policy
15	Smooth CNN	Apply mutation	22	Smooth mutation: MLP -> lightweight CNN via SmoothBlock(anchor=2,dur=8)
16	Smooth CNN	Complete training	41	Training phase completed

Tabella 10.2: Eventi chiave del caso di studio `alt-lineage-cnn-recovery`. La tabella proietta automaticamente dagli stream eventi la baseline comune, la derivazione dei due figli concorrenti e la seconda derivazione smooth verso CNN.

10.3.3 Metriche narrative e marker delle tre fasi smooth

Le curve di accuratezza e loss mostrano che il ramo hard raggiunge presto una CNN competitiva, ma il ramo di recupero smooth verso CNN riesce infine a superarlo leggermente sull'accuratezza di validazione e sulla loss di validazione finale. Nella versione `v3` dell'esperimento il trial *Hard CNN* termina con *val_acc* pari a 83.94% e *val_loss* pari a 0.4391, mentre il trial *Smooth CNN* chiude a 84.24% con *val_loss* pari a 0.4362. La baseline lineare e il passaggio smooth verso MLP restano importanti perché rendono leggibile la storia che porta a questo confronto finale, non solo il risultato numerico ultimo. La Figura 10.5 mostra questa dinamica sul piano dell'accuratezza, mentre la Figura 10.6 rende esplicita la genealogia completa dei trial coinvolti. La Figura 10.7 completa la lettura evidenziando anche i marker relative agli eventi delle transizioni architetturali.

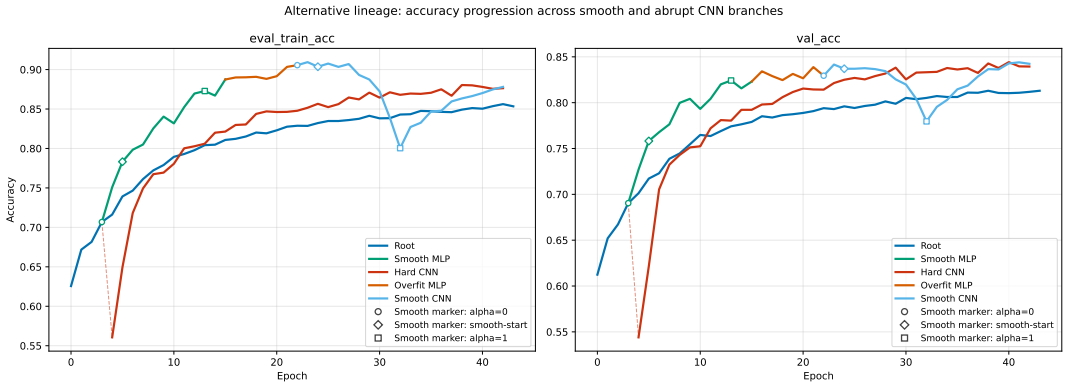


Figura 10.5: Curve di accuratezza del caso di studio `alt-lineage-cnn-recovery`. Il grafico mantiene la leggibilità genealogica della mutazione architetturale hard con un collegamento tratteggiato sottile e mostra i marker delle tre fasi delle transizioni smooth.

Figura 10.6: Grafo di derivazione completo del caso di studio `alt-lineage-cnn-recovery`. Il grafo rende leggibile la divergenza tra ramo smooth e ramo hard a partire dalla stessa baseline lineare.

L'esperimento rende espliciti anche i punti di transizione a tre stati della dinamica smooth. Per il trial *Smooth MLP* i marker principali cadono alle epoche 3, 5 e 13; per il trial *Smooth CNN* cadono alle epoche 22, 24 e 32. Questa lettura aiuta a separare tre fenomeni distinti: il tratto iniziale ancora equivalente al genitore, la fase di mescolamento progressivo tra `Old` e `New`, e la fase finale in cui la nuova architettura è ormai pienamente attiva. Dal punto di vista strutturale, la Figura 10.8 mette a confronto il ramo hard verso CNN e il ramo di recovery smooth verso CNN, mostrando che i due esiti finali derivano da storie architetturali diverse.

10.3.4 Che cosa dimostra rispetto al framework

Questo secondo caso dimostra soprattutto tre aspetti del framework. Primo, la derivazione di trial resta leggibile anche quando una baseline genera figli con semantica diversa, qui smooth e hard. Secondo, la politica dei pesi non è un dettaglio nascosto: il ramo hard parte davvero come trial figlio architetturalmente nuovo con reset dei pesi, mentre il ramo smooth conserva continuità parametrica controllata. Terzo, l'esperimento rende osservabili non solo eventi e metriche finali, ma anche stati intermedi della transizione smooth tramite marker narrativi coerenti con `anchor` e `duration` dichiarati nel protocollo sperimentale. La Tabella 10.3 riassume infine il confronto quantitativo tra baseline, ramo hard e ramo smooth di recovery, chiudendo la lettura con una sintesi compatta delle metriche finali.

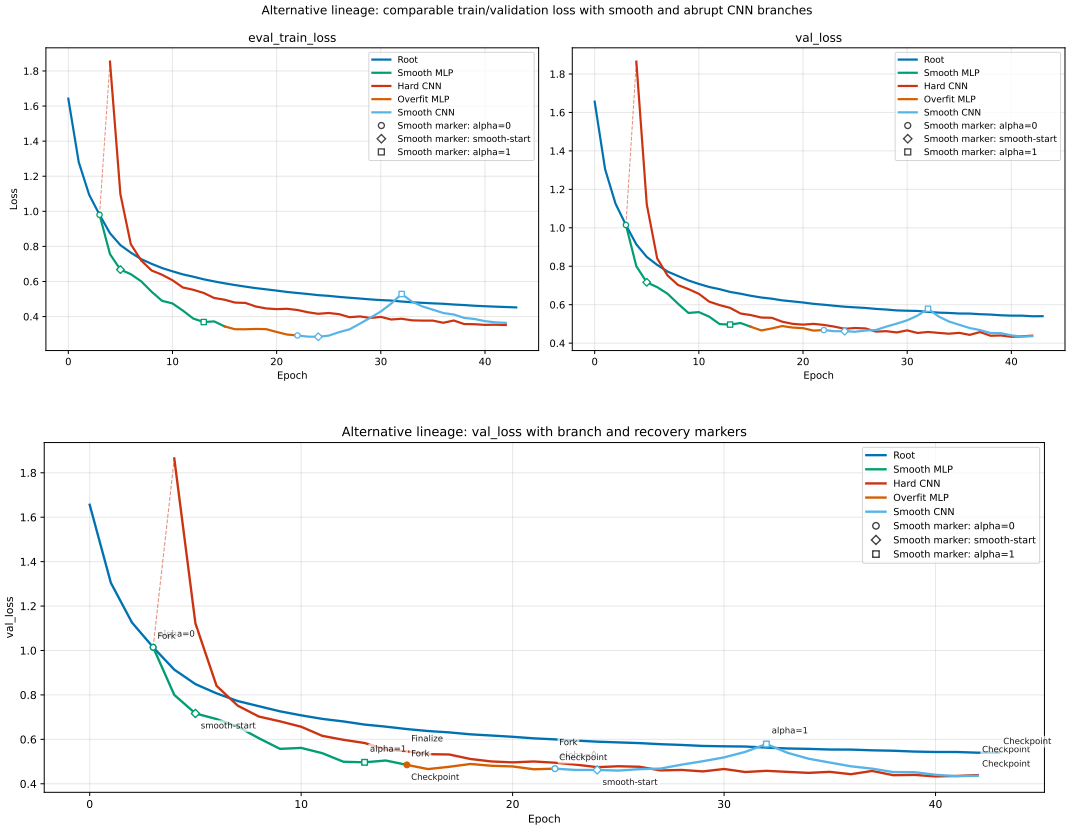
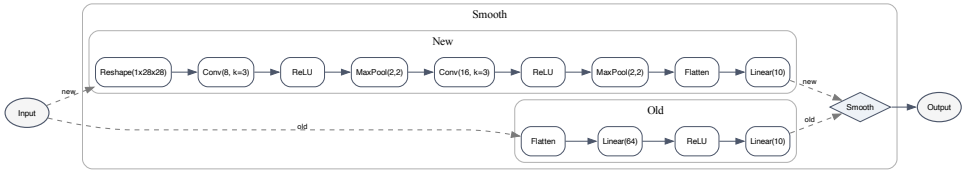


Figura 10.7: Curve di loss e marker di loss di validazione del caso di studio `alt-lineage-cnn-recovery`. Oltre ai marker finali, la seconda vista annota anche i tre stati delle due transizioni smooth: stato iniziale in `alpha=0`, inizio rampa e ingresso nello stato `alpha=1`.

(b) Ramo hard verso CNN con reset dei pesi



(c) Ramo di recupero smooth verso CNN con blocco `Old/New`

Figura 10.8: Snapshot strutturali del caso di studio `alt-lineage-cnn-recovery`. Il ramo hard verso CNN e la seconda transizione smooth che porta alla CNN di recupero.

Trial	Family	Epoch	Train loss	Val loss	Train acc	Val acc	Finalized
Root	linear	43	0.455	0.540	0.855	0.813	no
Smooth MLP	smooth	15	0.372	0.485	0.873	0.823	no
Hard CNN	cnn	42	0.358	0.439	0.881	0.839	no
Overfit MLP	mlp	22	0.303	0.468	0.897	0.829	yes
Smooth CNN	smooth	42	0.372	0.436	0.875	0.842	no

Tabella 10.3: Sintesi finale delle metriche del caso di studio `alt-lineage-cnn-recovery`. La tabella è generata automaticamente dai report in formato CSV dell'esperimento e rende confrontabili baseline, ramo hard, ramo smooth intermedio e recupero finale.

Elemento	Valore usato nell'esperimento
Script di lancio	`scripts/run_thesis_case_study_3.sh`
Scenario base	`demo-fast -device cpu -no-hard-arch`
Parametri principali	`initial_epochs=10`, `parent_epochs=3`, `candidate_epochs=16`, `smooth_duration=4`
Procedura di ricerca	`rounds=3`, `children=3`, selezione su `val_acc`
Scelte architetturali	`arch-hidden-linear=128`, `mlp-to-cnn-batch=256`, `cnn-to-vgg-hidden=64`
Scelte HPO	`default-lr=0.001`, `lr-choices={0.0001, 0.0002}`, `batch-choices={128}`

Tabella 10.4: Configurazione operativa del caso di studio `case_study_3_progressive_search_v1`. La tabella riassume lo script di lancio e i parametri che fissano il budget della procedura di ricerca multi-round.

10.4 Caso di studio 3: ricerca multi-round a partire da una baseline comune

Il terzo caso di studio usa l'esperimento `case_study_3_progressive_search_v1`, costruito per mostrare una storia di ricerca compatta ma non banale a partire da una baseline lineare comune. L'obiettivo qui non è isolare una singola mutazione, ma rendere leggibile una procedura di ricerca multi-round in cui il trial selezionato di ogni round diventa il genitore del round successivo e la scelta avviene rispetto alla vista osservazionale dell'accuratezza di validazione.

10.4.1 Scopo e configurazione della procedura di ricerca

La configurazione resta intenzionalmente leggera per contenere il costo dell'esperimento: una baseline lineare iniziale addestrata per 10 epoche, tre round di selezione, tre figli per round, continuazione del genitore selezionato per 3 epoche prima di generare i nuovi candidati, e training di 16 epoche per ciascun figlio. In ogni round il framework genera un candidato con mutazione smooth architetturale e due candidati con variazioni iperparametriche; il vincitore è selezionato in base a `val_acc` e prosegue da solo nel round successivo. La Tabella 10.4 riassume questi parametri operativi e fissa il budget entro cui leggere i risultati del caso di studio.

10.4.2 Grafo di derivazione della ricerca e selezione round-by-round

La storia risultante è più ricca del primo caso, ma resta compatta abbastanza da poter essere letta come una singola traiettoria sperimentale. Il trial radice lineare raggiunge un'accuratezza di validazione pari a 79.40% alla fine della fase iniziale; nel primo round vince il figlio smooth `r1_c01_smooth_arch`, che materializza la transizione lineare → MLP e porta la prestazione a 84.26%. Quel trial viene poi finalizzato e riusato come genitore del round successivo, dove questa volta il confronto rispetto alla vista delle metriche seleziona ancora il candidato architetturale: `r2_c01_smooth_arch` realizza la transizione MLP → CNN e porta l'accuratezza di validazione a

Round	Trial selezionato	Mutazione prevalente	Acc. di validazione
0	`search_parent_r0`	init-linear	79.40%
1	`r1_c01_smooth_arch`	smooth-linear-to-mlp-h128	84.26%
2	`r2_c01_smooth_arch`	smooth-mlp-to-cnn-lr0.001-b256	85.52%
3	`r3_c03_hyper`	hyper-lr0.0002-b128	86.48%

Tabella 10.5: Storia dei trial selezionati della procedura di ricerca multi-round. La sequenza mostra una progressione monotona del trial selezionato rispetto alla vista `val_acc`.

85.52%. Nel terzo round il genitore CNN viene continuato e confrontato sia con una nuova mutazione smooth verso VGG-lite sia con due raffinamenti iperparametrici; il vincitore `r3_c03_hyper` resta nella famiglia CNN e spinge l'accuratezza di validazione fino a 86.48%.

Questo andamento rende esplicita una proprietà importante del framework: la mutazione architetturale non viene privilegiata a priori, ma può davvero emergere come vincitore quando il budget del round è sufficiente a far maturare il cambio di famiglia. La Tabella 10.5 rende immediata questa progressione del vincitore round-by-round. Il grafo di derivazione conserva il doppio punto in cui la storia cambia famiglia architetturale, prima da lineare a MLP e poi da MLP a CNN, mentre la selezione round-by-round resta una decisione osservazionale fondata sulle metriche persistite nell'esperimento.

Come su può osservare nella Tabella 10.4, la lettura dell'esperimento può essere resa esplicita anche a livello di storia degli eventi selezionati. Invece di riportare l'intero archivio di eventi di tutti i candidati, qui conviene mostrare la traiettoria che struttura davvero la narrazione del caso di studio: il trial radice e i tre trial selezionati che fissano la progressione dei round.

10.4.3 Metriche narrative e artefatti prodotti

L'esperimento include un grafico narrativo che sovrappone `train_acc` e `val_acc` lungo il percorso dei trial rilevanti, lasciando sullo sfondo i candidati non selezionati come contesto del confronto. Accanto a questo, una seconda vista sintetizza la sola storia dei vincitori round-by-round. La Figura 10.9 mostra il contesto completo dei trial selezionati e non selezionati, mentre la Figura 10.11 isola la sola traiettoria dei vincitori. Il corrispondente grafo di derivazione completo è riportato in Figura 10.10. Le due figure non introducono uno stato alternativo: sono proiezioni rigenerabili della stessa storia degli eventi e degli stessi report CSV prodotti dal flusso di lavoro.

L'esperimento non contiene un trial VGG-lite finale selezionato o finalizzato: nel terzo round il candidato architetturale verso VGG-lite viene generato e persistito come trial smooth, ma il confronto rispetto alla vista delle metriche sceglie invece il ramo CNN iperparametrico. Per questo motivo la figura strutturale più informativa non è una ipotetica architettura finale VGG-lite, ma la coppia composta dal genitore CNN selezionato e dal trial smooth in cui il target VGG-lite compare nel ramo `New`. Questa lettura è resa esplicita in Figura 10.12, che mette a confronto il genitore selezionato al round 2 e il candidato smooth verso VGG-lite generato nel round 3.

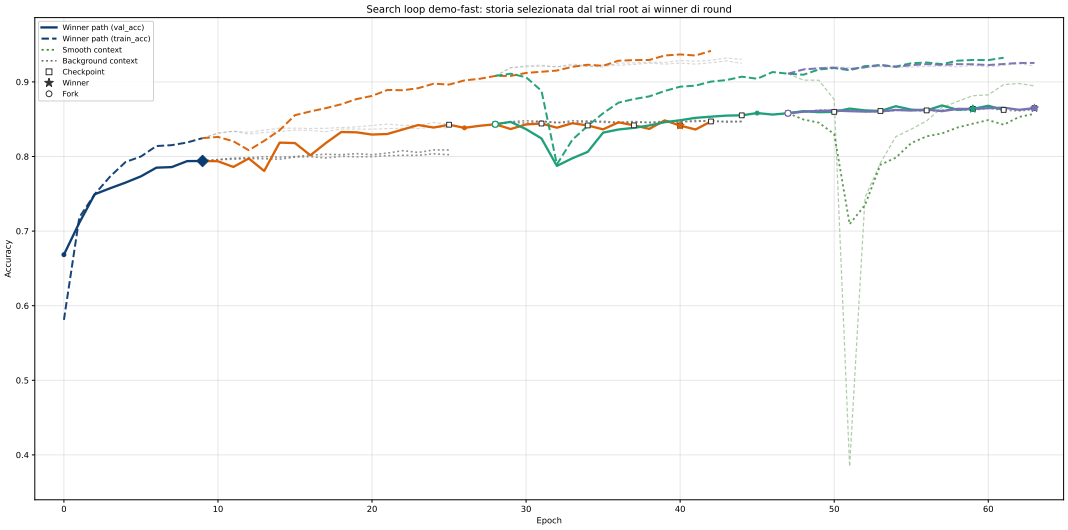


Figura 10.9: Curva narrativa del caso di studio multi-round. Il grafico mostra la traiettoria del trial radice e dei trial selezionati, mantenendo i candidati non scelti come contesto attenuato del confronto.

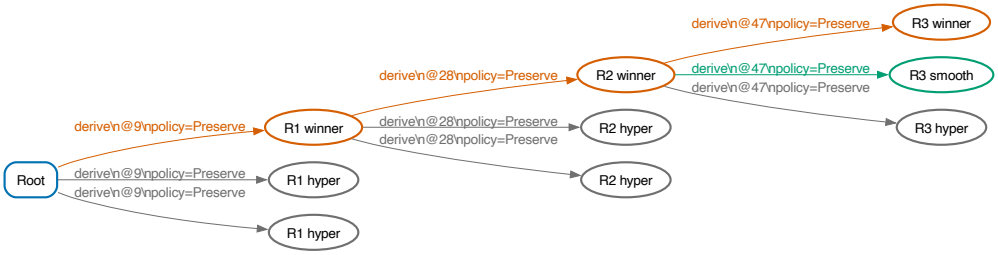


Figura 10.10: Grafo di derivazione completo del caso di studio multi-round. Oltre alla traiettoria dei trial selezionati, il grafo mantiene visibili anche i candidati smooth e iperparametrici non selezionati, in modo che la decisione di ogni round resti leggibile come confronto dentro uno stesso albero di derivazione.

10.4.4 Che cosa dimostra rispetto al framework

Questo terzo caso dimostra che il framework riesce a sostenere una ricerca multi-round riproducibile senza perdere leggibilità narrativa. Primo, la derivazione di trial resta esplicita: ogni round introduce nuovi figli con identità propria e un genitore ben definito. Secondo, la selezione del vincitore resta giustificabile come confronto rispetto a una vista osservazionale concreta, qui l'accuratezza di validazione, invece di dipendere da euristiche opache esterne all'esperimento. Terzo, il flusso di lavoro produce artefatti abbastanza stabili da supportare insieme replay, sequenze eventi selezionate, grafo di derivazione completo, report CSV, grafici narrativi e inclusione diretta in tesi.

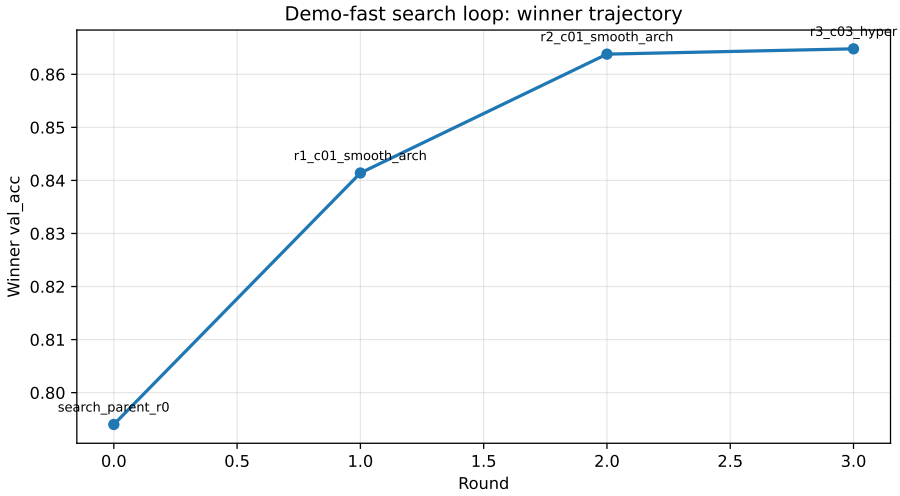
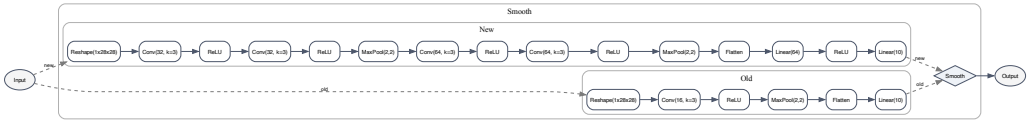


Figura 10.11: Sintesi dei trial selezionati della procedura di ricerca. La vista rende immediato il miglioramento cumulativo del trial selezionato nei tre round.

(a) Genitore CNN selezionato al round 2



(b) Trial smooth del round 3: ingresso comune, rami `Old`/`New` e merge esplicito in `Smooth`

Figura 10.12: Artefatti strutturali del caso di studio multi-round. Il primo pannello mostra il genitore CNN che entra nel terzo round; il secondo mostra la mutazione smooth verso VGG-lite: ingresso comune dal nodo di input, due cammini paralleli `Old`/`New` e nodo `Smooth` come punto di fusione prima dell'uscita.

No.	Trial	Action	Epoch	Summary
1	Root	Create trial		Trial created
2	Root	Apply mutation	0	Bootstrap canonical linear baseline from MlProfileBuilders
3	Root	Complete training	9	Training phase completed
4	R1 winner	Derive trial	9	Derived from Root with preserve policy
5	R1 winner	Apply mutation	9	Smooth mutation: linear -> MLP(hidden=[128]) via SmoothBlock(anchor=0,dur=4)
6	R1 winner	Complete training	24	Training phase completed
7	R1 winner	Apply mutation	26	Finalize smooth: collapse all SmoothBlocks to New branch [mature finalize: alpha=AlphaEnd; exact continuity expected]
8	R1 winner	Complete training	28	Training phase completed
9	R2 winner	Derive trial	28	Derived from R1 winner with preserve policy
10	R2 winner	Apply mutation	28	Smooth mutation: MLP -> CNN baseline via SmoothBlock(anchor=0,dur=4)
11	R2 winner	Complete training	43	Training phase completed
12	R2 winner	Apply mutation	45	Finalize smooth: collapse all SmoothBlocks to New branch [mature finalize: alpha=AlphaEnd; exact continuity expected]
13	R2 winner	Complete training	47	Training phase completed
14	R3 winner	Derive trial	47	Derived from R2 winner with preserve policy
15	R3 winner	Apply mutation		round 3 hyper candidate: lr=0.0002, batch=128
16	R3 winner	Complete training	62	Training phase completed
17	R1 winner	Complete training	31	Training phase completed
18	R2 winner	Complete training	50	Training phase completed
19	R1 winner	Complete training	34	Training phase completed
20	R2 winner	Complete training	53	Training phase completed
21	R1 winner	Complete training	37	Training phase completed
22	R2 winner	Complete training	56	Training phase completed
23	R1 winner	Complete training	40	Training phase completed
24	R2 winner	Complete training	59	Training phase completed
25	R1 winner	Complete training	42	Training phase completed
26	R2 winner	Complete training	61	Training phase completed

Tabella 10.6: Sequenza di eventi chiave del caso di studio `case_study_3_progressive_search_v1`. La tabella proietta automaticamente dagli stream eventi il percorso Root → R1 trial vincitore → R2 trial vincitore → R3 trial vincitore.

10.5 Caso di studio 4: trasformazione strutturale MADNet-like per stereo matching

Il quarto caso di studio ha uno scopo diverso rispetto ai tre casi precedenti. Qui non interessa discutere l'addestramento di un modello, ma mostrare che il framework riesce a produrre artefatti leggibili anche quando il trial resta deliberatamente *puramente strutturale*. Il punto non è quindi una prestazione osservata su metriche di training, ma la tenuta del percorso concettuale che va dal trial al bigrafo, inteso qui come struttura che combina contenimento e connettività, dal bigrafo alla vista strutturale, e da questa agli artefatti DOT, al grafo di derivazione e ai report.

Il caso é ispirato a una famiglia di pipeline stereo coarse-to-fine vicine al lessico MADNet, ma é ridotto intenzionalmente a un confronto strutturale controllato. La riduzione é severa: nessun training reale, nessun benchmark, nessun tentativo di riprodurre integralmente un modello stereo dello stato dell'arte. Restano invece di primo piano tre elementi che la tesi vuole rendere verificabili: una radice architetturale leggibile in stile MADNet-like, una mutazione architetturale smooth che sostituisce localmente l'ultimo blocco di raffinamento, e un figlio finale che mostra l'effetto di `finalize-smooth` come collasso della mutazione locale nella catena principale.

10.5.1 Scopo del caso e perimetro dichiarato

Questo caso serve a verificare che il framework possa estendere il proprio lessico strutturale verso architetture stereo multistadio in stile MADNet-like senza aprire subito un backend eseguibile dedicato. Per questo la mutazione architetturale é trattata qui come fatto semantico e come storia di trial ricostruibile tramite replay, non come training effettivo di un modello stereo. I tre trial dell'esperimento restano comunque trial completi nel senso della tesi: hanno identità, grafo di derivazione minimo, storia degli eventi e artefatti derivati, ma lasciano volutamente vuota la parte osservazionale legata a metriche di addestramento che richiederebbe un supporto esecutivo ancora da sviluppare.

La forma scelta per l'architettura radice non é più una semplice rete lineare, ma una decomposizione più vicina al lessico MADNet-like: uno `StereoStem` con località `Left`, `Right` e `Fuse`, un `MatchStage`, due blocchi di raffinamento distinti e un `DisparityHeadBlock` finale. Questa scelta rende la narrativa strutturale più simile a una rete stereo classica: il confronto tra viste entra presto nel grafo, il raffinamento é esplicito come sequenza di blocchi, e la mutazione smooth può essere agganciata a un punto locale semanticamente chiaro.

10.5.2 Grafo di derivazione minimo: radice, figlio smooth, figlio finalizzato

Il grafo di derivazione dell'esperimento é intenzionalmente minimale. Il trial radice crea la struttura MADNet-like di riferimento; il figlio *Smooth* introduce un `SmoothBlock` che intercetta il secondo blocco di raffinamento e conserva nel ramo `Old` la versione radice di `RefineStage B`, mentre nel ramo `New` materializza una

Trial	Label	Stages	Smooth	Cost volume	Head	Nodes	Links
exp04_madnet_root	Root	5	0	1	1	114	20
exp04_madnet_child_smooth	Smooth	6	1	1	1	145	24
exp04_madnet_child_finalized	Finalized	5	0	1	1	118	22

Tabella 10.7: Sintesi strutturale dei tre trial dell'esperimento. Il figlio smooth rende esplicita la mutazione locale su `RefineStage B`, mentre il figlio finale mostra il collasso della transizione smooth in una nuova catena strutturale.

No.	Trial	Action	Epoch	Summary
1	Root	Apply mutation		EXP-04 MADNet-like structural root
2	Smooth	Derive trial	0	Derived from Root with preserve policy
3	Smooth	Apply mutation	0	EXP-04 MADNet-like smooth local replacement
4	Finalized	Derive trial	0	Derived from Smooth with preserve policy
5	Finalized	Apply mutation	0	EXP-04 MADNet-like finalize-smooth

Tabella 10.8: Sequenza dei passi di mutazione dell'esperimento. La tabella mostra la creazione della radice, la derivazione smooth locale e il passo di `finalize-smooth`, con i rispettivi eventi di dominio e le descrizioni sintetiche.

variante più ricca con combinazione residua esplicita. Il terzo trial applica `finalize-smooth` e collassa la transizione locale nel ramo `New`, ottenendo una vista finale in cui la mutazione non é più rappresentata come nodo smooth ma come nuova catena canonica del trial figlio.

Questa lettura é coerente con la semantica del framework. La derivazione non é presentata come copia informale di un grafo, ma come nuova identità di trial ricostruibile via replay, con `TrialForked`, `ForkSnapshot` e `PatchApplied` persistiti nel rispettivo stream eventi. L'esperimento é quindi utile non solo come raccolta di artefatti DOT, ma come micro-caso di grafo di derivazione osservabile in cui il cambiamento architetturale resta esplicito e confrontabile. La Tabella 10.7 sintetizza i tre trial dal punto di vista della struttura, mentre la Tabella 10.8 rende esplicita la sequenza dei passi canonici che porta dalla radice al trial finalizzato, con i rispettivi eventi di dominio e descrizioni sintetiche.

10.5.3 Artefatti prodotti e loro ruolo nel capitolo

Dal punto di vista degli artefatti, questo esperimento é importante perché sposta l'attenzione dalle metriche alle proiezioni strutturali. Il framework materializza per ciascun trial una versione DOT estesa, una collassata, un export `.big` coerente con la vista dati, il DOT complessivo del grafo di derivazione e due report di differenza, radice→smooth e smooth→finalized, che rendono leggibile il cambiamento come fatto esplicito della storia.

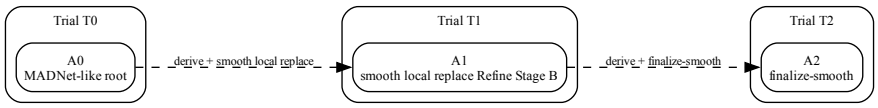


Figura 10.13: Grafo di derivazione dell'esperimento. Il trial radice viene derivato in Smooth, che a sua volta viene derivato in Finalized. La mutazione smooth locale e il passo di finalize-smooth sono evidenziati come eventi di dominio chiave.

In questo caso il valore dell'esperimento non sta in una curva di training inesistente, ma nel fatto che l'insieme di riferimenti ad artefatti, DOT e tabelle nasce in modo rigenerabile da una storia replicabile e rende leggibili quattro viste distinte: architettura radice estesa, mutazione smooth locale, architettura finale collassata e grafo di derivazione minimale del trial. La Figura 10.13 mostra il grafo di derivazione minimo dell'esperimento, mentre la Figura 10.14 rende visibili le tre viste strutturali che corrispondono alla radice, al trial smooth dopo una riscrittura sulla radice ed al trial finalizzato dopo la mutazione dello smooth.

10.5.4 Che cosa aggiunge rispetto agli altri casi di studio

I primi tre casi di studio del capitolo mostrano esperimenti maturi nel perimetro eseguibile già consolidato del framework: training reale, metriche osservate, grafici e selezione rispetto a una vista di metrica. Questo quarto caso aggiunge invece un altro tipo di maturità: mostra che il framework sa già sostenere una trasformazione strutturale credibile anche oltre il profilo eseguibile attuale, purché il perimetro semantico resti dichiarato con rigore.

Per la tesi questo punto conta molto. Questo esperimento non dimostra ancora capacità di training stereo, ma dimostra che i concetti di trial, mutazione architetturale, grafo di derivazione, replay e confronto di artefatti non dipendono strettamente dal perimetro TorchSharp oggi eseguibile. In altre parole, il framework riesce già a produrre evidenza tecnica leggibile per un caso stereo MADNet-like senza confondere supporto strutturale e supporto esecutivo completo.

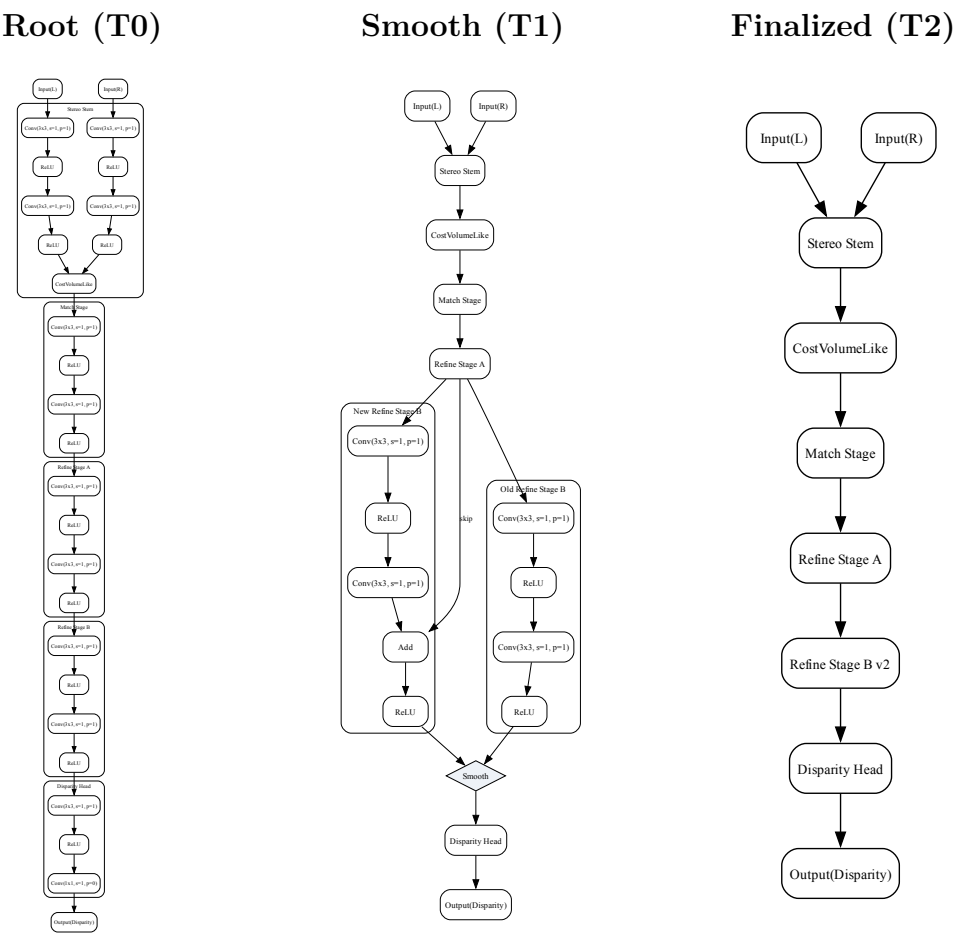


Figura 10.14: Viste delle due trasformazioni architetturali dell'esperimento, disposte da sinistra a destra come (i) radice (vista di dettaglio), (ii) mutazione locale smooth (vista di dettaglio) e (iii) architettura finalizzata (vista d'insieme).

10.6 Conclusione sui casi di studio

Rispetto a Capitolo 9, questo capitolo non valuta proprietà ma mostra prodotti sperimentali concreti del framework: timeline selettive, curve, grafi di derivazione, snapshot strutturali e tabelle finali compatte. Letti insieme, i casi di studio qui raccolti mostrano che questa capacità non si limita a un solo tipo di storia sperimentale: il framework riesce a rendere osservabile sia una derivazione con rami concorrenti smooth e hard, sia una derivazione controllata con finalizzazione, sia una ricerca multi-round in cui la selezione dei trial migliori, il grafo di derivazione e i report restano leggibili come parti coerenti dello stesso esperimento, sia infine un caso puramente strutturale in cui gli artefatti restano coerenti anche senza training reale. Rispetto a Capitolo 11, non discute ancora limiti o sviluppi futuri: prepara piuttosto il terreno mostrando quali esperimenti risultano già abbastanza maturi da sostenere una narrazione tecnica leggibile e quindi da fondare la discussione finale su evidenze concrete, non solo su proprietà astratte.

Il quadro che emerge è coerente con la tesi complessiva. Il framework non si limita a registrare una storia di eventi: produce artefatti derivati abbastanza stabili da sostenere replay, confronto, ispezione manuale e documentazione scientifica all'interno dello stesso flusso di lavoro, e proprio per questo rende naturale il passaggio dal piano della valutazione al piano della discussione finale.

Capitolo 11

Discussione e lavori futuri

Questo capitolo discute i compromessi progettuali del framework proposto, i limiti attuali e le principali direzioni di sviluppo. La discussione è organizzata attorno ai pilastri della tesi: IR basata su bigrafo e riscrittura, tracciabilità basata su *event sourcing* con replay ed eventi di dominio arricchiti da metadati di provenienza, interfaccia verso l'esterno (*boundary*) verso strumenti esterni, e controller intercambiabile (manuale o automatico). [7], [8], [13], [18]

11.1 Sintesi dei contributi

La tesi propone un framework per *Automated Machine Learning* (AutoML) tracciabile, oggi centrato soprattutto su ricerca dell'architettura (*neural architecture search*, NAS) e ottimizzazione degli iperparametri, basato su *event sourcing* e su riscrittura di bigrafi. In sintesi, i contributi sono:

- una modellazione esplicita del trial come unità primaria del framework, con stato canonico descritto da $T = \langle A, H, W, P, O, L \rangle$ e con $\text{core}(T) = \langle A, H, W \rangle$ come proiezione leggera;
- l'uso di bigrafi e *Bigraphical Reactive Systems* (BRS) come IR per mutazioni architetturali (riscrittura su grafi); [13]
- una semantica a eventi con replay deterministico e record di evento che separano metadati di provenienza e payload di dominio;
- una nozione di confronto rispetto a una vista, usata per rendere confrontabili i cambiamenti osservabili;
- interfaccie per un controller intercambiabile (manuale o automatico) che consente di passare dalla gestione manuale a strategie automatizzate senza modificare il core.

11.2 Compromessi progettuali

Tracciabilità vs overhead. Passi canonici registrati e metadati di provenienza introducono overhead di tempo e spazio. Il framework adotta un compromesso: il

log resta leggero (eventi + riferimenti ad artefatti immutabili), mentre i dati pesanti (checkpoint, curve metriche, viste esterne) sono delegati ad artefatti indicizzati e versionati. In questo modo si può arricchire il reporting (nuove viste o nuovi read model) senza modificare la semantica del core.

Generalità dell'IR vs semplicità. I bigrafi sono un formalismo generale; qui sono usati principalmente come IR dell'architettura, con un linguaggio di riscrittura che rimane localmente verificabile. Il vantaggio è la controllabilità dello spazio di trasformazioni e la possibilità di validazione strutturale; lo svantaggio è l'aumento dei vincoli da mantenere (signature, typing, well-formedness) e del tooling necessario. [13]

Trasformazioni discrete vs continuità del training. Una riscrittura rende esplicita la mutazione, ma non garantisce continuità del training. Il framework consente di esprimere strategie di transizione (restart, avvio a caldo o *warm-start*, trasferimento di pesi), ma resta necessario scegliere quali transizioni siano appropriate per un dato dominio. Questo tema è legato sia a vincoli di forma (shape/typing) sia a proprietà della trasformazione (quanto preserva la funzione).

Boundary verificabile vs fiducia nello strumento esterno. Il boundary consente roundtrip controllati e import compatibili con il replay, ma gli strumenti esterni possono evolvere, cambiare formato o produrre output non validi. La robustezza dipende dalla qualità degli adattatori (parsing/normalizzazione) e dalla copertura degli invarianti di rientro; i metadati di provenienza riducono l'ambiguità ma non eliminano la necessità di validazione.

11.3 Limiti attuali

Per evitare che i limiti restino "narrativi", li riassumiamo con impatto e mitigazioni previste nella Tabella 11.1.

11.4 Lavori futuri

Controller automatico (AutoML). Il passo più naturale è sostituire progressivamente la gestione manuale con strategie automatizzate: random search, bandit ed *early stopping*, ottimizzazione bayesiana, e politiche basate sul grafo delle transizioni. [3], [12], [22] Il design a eventi consente di confrontare policy diverse a posteriori senza modificare il dominio.

Transizioni smooth e morphing. Un'area ricca è rendere più sistematiche le transizioni tra architetture per ridurre instabilità delle metriche: trasferimento di pesi, gating e morphing graduale, con vincoli che preservano (quando possibile) relazioni tra funzioni rappresentate. Nel framework queste tecniche si esprimono come sequenze controllate di riscritture e aggiornamenti di configurazione o training, mantenendo tracciabilità e confrontabilità tramite confronti rispetto a una vista.

Limite	Impatto	Mitigazione / estensione
Valutazione orientata a proprietà	Non misura prestazioni di stato dell'arte; misura correttezza/robustezza del framework	Estendere i casi d'uso con dataset e budget maggiori, mantenendo gli stessi oracoli (E1–E4) [16]
Copertura trasformazioni parziale	Catalogo regole incompleto; rischio di regressioni su nuove mutazioni	Aggiungere regole + test generativi e report di regressione per confronti rispetto a una vista
Determinismo del training non garantito	HW/librerie possono introdurre non-determinismo end-to-end	Registrare seed/versioni; usare confronti su artefatti verificabili e tolleranze dichiarate [16]
Maturità degli adattatori (import/export)	Interoperabilità limitata verso ecosistemi esterni	Adattatori incrementali (ONNX, PyTorch), con roundtrip e validazione strutturale

Tabella 11.1: Limiti attuali: impatto e mitigazioni. La tesi privilegia proprietà verificabili rispetto a benchmark prestazionali. [20]

Asincronia, parallelismo e orchestrazione job. Pur mantenendo un core deterministico, è possibile estendere l'orchestrazione per supportare esecuzioni concorrenti: job queue, batching di richieste e completamenti fuori ordine, e read model dei job attivi. Questa estensione richiede politiche idempotenti ben definite e test di replay più estesi.

Estensione dell'IR e viste "paper-ready". Due direzioni complementari sono: (i) estendere l'IR basata su bigrafo oltre l'architettura (processo, dati, strumenti) per modellare porzioni più ampie del sistema; (ii) arricchire le viste (DOT/bigrafo) con stili configurabili e export coerenti con figure da pubblicazione.

Interoperabilità con ecosistema ML. L'integrazione di backend aggiuntivi e adattatori di import/export (es. ONNX) favorirebbe l'adozione del framework. Un approccio incrementale è introdurre adattatori che traducono modelli esterni in IR basata su bigrafo, preservando identità e vincoli dove possibile.

Formalizzazione più forte. È possibile rafforzare la parte formale: leggi e proprietà per il boundary ispirato alle lenses, oppure modelli composizionali per le trasformazioni e per la composizione degli eventi di dominio. Questa tesi adotta una formalizzazione leggera e verificabile; sviluppi futuri potrebbero spingersi verso risultati più strutturati.

11.5 Conclusioni

La tesi mostra come combinare riscrittura di bigrafi, *event sourcing* e un boundary verificabile per costruire un'infrastruttura AutoML tracciabile. Il risultato è un

framework che rende esplicite le trasformazioni, ricostruibile la traiettoria e intercambiabile la sorgente delle decisioni (manuale o automatica). Queste proprietà forniscono una base solida per ricerca sperimentale, debugging e sviluppo di controller AutoML più avanzati, mantenendo separati semantica del dominio ed effetti esterni. [20]

Appendice A

Convenzioni, terminologia e notazione

Questa appendice fissa la notazione usata nell'elaborato e disambigua concetti che possono essere letti in modi diversi. In particolare, separa: (i) il *modello canonico del dominio*, (ii) la grammatica primaria `command / event / decide-evolve`, (iii) le nozioni derivate di *passo canonico*, *confronto rispetto a una vista*, *patch* e *reaction rule*. Questa appendice adotta come riferimento il modello concettuale presentato nella tesi; il prototipo attuale ne realizza una porzione consistente, ma non costituisce la base definitoria di questa sezione.

A.1 Terminologia

La Tabella A.1 raccoglie i termini ricorrenti nell'elaborato e la loro interpretazione nel framework.

A.2 Oggetti principali

Esperimento e trial

Nel seguito usiamo come forma canonica dell'esperimento:

$$E := \langle C, \mathcal{T} \rangle$$

dove C raccoglie il contesto condiviso dell'esperimento (ad esempio dataset, protocollo o assunzioni comuni) e $\mathcal{T}$ è l'insieme dei trial appartenenti a quello spazio sperimentale.

L'unità sperimentale primaria è il trial:

$$T := \langle A, H, W, P, O, L \rangle$$

dove:

- A è l'architettura;
- H sono gli iperparametri;

- W rappresenta lo stato parametrico del trial; nel prototipo esso può essere materializzato o referenziato tramite checkpoint e artefatti del backend;
- P è lo stato di processo;
- O è l'insieme delle osservazioni rilevanti per analisi e confronto;
- L è il lineage minimo del trial, ossia il suo grafo di derivazione minimo.

Questa notazione descrive il *modello concettuale canonico del dominio*, non una trascrizione letterale dei tipi concreti del prototipo.

Proiezioni leggere del trial

Quando serve parlare del nucleo del trial senza portare con sé tutto il contesto sperimentale, usiamo:

$$\text{core}(T) := \langle A, H, W \rangle$$

Quando invece serve parlare del modello in senso stretto, usiamo la vista derivata:

$$M(T) := \langle A, W \rangle$$

Di conseguenza:

- il trial completo resta l'entità primaria del dominio;
- il *core* è una proiezione espositiva del trial;
- il *modello* $M(T)$ è una vista derivata utile in alcuni passaggi della tesi, ma non sostituisce il trial come oggetto fondativo.

Architettura, osservazioni e lineage

L'architettura A è modellata come un bigrafo, vale a dire una rappresentazione che combina struttura di annidamento e connettività. Questa scelta è forte per la componente architetturale del trial, ma non implica che l'intero framework sia ridotto a un sistema di riscrittura bigrafica. [1], [13], [14], [21]

Le osservazioni O raccolgono ciò che serve per leggere e confrontare il comportamento sperimentale del trial. In particolare, nel modello canonico includono almeno:

- metriche;
- checkpoint;
- marker di fase;
- provenance utile a spiegare alcuni cambiamenti.

Il lineage, ossia il grafo di derivazione minimo del trial, è:

$$L := \langle id, parent, src \rangle$$

dove id identifica il trial, $parent$ denota eventualmente il trial padre e src fissa la sorgente immediata della derivazione. Provenance più ricche possono vivere in eventi, viste e artefatti, ma non fanno parte del nucleo minimale di L.

A.3 Grammatica primaria: comandi, eventi e replay

La semantica primaria del framework non è fondata su una singola freccia del tipo $T \xrightarrow{\delta} T'$, ma sulla grammatica:

- `command`: richiesta o intenzione;
- `event`: fatto registrato nella storia;
- `decide`: traduzione di una richiesta in una lista finita di eventi ammessi;
- `evolve`: applicazione di un singolo evento allo stato.

In forma astratta, useremo:

$$\text{decide} : S \times Q \to Ev^*$$

$$\text{evolve} : S \times e \to S$$

dove S denota lo stato rilevante del dominio, Q lo spazio dei command, e un evento singolo e Ev^* una lista finita di eventi.

Il replay è quindi un fold sugli eventi registrati. Per un log $\bar{e} = [e_1, \ldots, e_n]$ scriveremo:

$$\text{replay}(S_0, \bar{e}) := \text{evolve}^*(S_0, \bar{e})$$

Nel prototipo attuale, la fonte di verità resta la storia append-only degli eventi; le viste e i report sono derivazioni rigenerabili da tale storia. [8]

A.4 Passo canonico come nozione derivata

Il termine *passo canonico* resta utile, ma nel modello attuale va letto come *nozione derivata*, non come primitivo fondativo della semantica.

Forma generale: livello di esperimento

La forma più generale e consigliata è definita a livello di esperimento:

$$\delta_q^E(E) := \text{evolve}^*(E, \text{decide}(E, q))$$

Questa forma è la più naturale per famiglie di cambiamento come:

- `create`, che introduce un nuovo trial radice;
- `derive`, che introduce un nuovo trial figlio;
- `train`, `proc`, `obs`, che possono comunque essere letti come trasformazioni dell'esperimento che contiene i trial.

Forma locale: livello trial

Quando invece un comando è naturalmente locale a un trial già esistente, si può usare anche la forma definita a livello di trial:

$$\delta_q^T(T) := \text{evolve}^*(T, \text{decide}(T, q))$$

Questa forma è utile soprattutto per workflow come `train`, `proc` e `obs`, ma non è la forma più naturale per `create` e `derive`.

Variante event-first

Quando il passo viene già descritto come lista di eventi, useremo anche:

$$\delta_{\bar{e}}^E(E) := \text{evolve}^*(E, \bar{e}) \qquad \text{con} \qquad \bar{e} = [e_1, \ldots, e_n]$$

oppure, localmente sul trial:

$$\delta_{\bar{e}}^T(T) := \text{evolve}^*(T, \bar{e})$$

Conseguenze terminologiche

Nel testo della tesi vale quindi la seguente convenzione:

- la grammatica primaria resta `command / event / decide-evolve`;
- il simbolo δ resta ammesso come scorciatoia o metavariabile astratta;
- quando serve massima fedeltà al canone del framework, conviene esplicitare direttamente `decide` ed `evolve` invece di comprimere tutto in una singola freccia.

A.5 Confronti rispetto a una vista

Il confronto è una nozione *derivata e osservazionale*. Non muta lo stato del trial e non coincide con il passo canonico.

Vista

Useremo come forma canonica della vista:

$$V : \mathcal{T} \to Y_V$$

Dove $\mathcal{T}$ è l'insieme dei trial e Y_V è il codominio della vista scelta. Una vista prende un trial e ne estrae una rappresentazione confrontabile.

Operatore di confronto

La forma canonica del confronto sotto vista è:

$$\Delta_V : \mathcal{T} \times \mathcal{T} \to D_V$$

$$\Delta_V(T_1, T_2) := \mathrm{diff}_V\Big(V(T_1), V(T_2)\Big)$$

Dove D_V è il tipo delle differenze strutturate per la vista V.

Importante: Δ_V non restituisce necessariamente uno scalare. Nel quadro di questa tesi va pensato come una *differenza strutturata*, non come una distanza numerica.

Equivalenza osservazionale

Definiamo inoltre l'equivalenza sotto vista:

$$T_1 \equiv_V T_2 \iff \Delta_V(T_1, T_2) = \varnothing_V$$

dove $\varnothing_V$ denota la differenza vuota per la vista V.

Viste canoniche candidate

Le viste minime più naturali nel quadro della tesi sono:

$$V_A(T) = A, \quad V_H(T) = H, \quad V_W(T) = W, \quad V_P(T) = P, \quad V_O(T) = O, \quad V_L(T) = L,$$

$$V_{\mathrm{core}}(T) = \mathrm{core}(T), \qquad V_M(T) = M(T) = \langle A, W \rangle.$$

A.6 Patch, reaction e mutazione architetturale

Nel caso delle trasformazioni architetturali, il framework distingue tre livelli:

- **reaction rule**: schema generativo o spiegazione operazionale del cambiamento di A;
- **patch**: payload tecnico concreto di una mutazione architetturale;
- **evento canonico di dominio**: fatto registrato nella storia del trial.

Formula minima

Nel caso reaction-driven, la lettura più utile è:

$$\mathrm{compile}(r, m, A) = p$$

e poi:

$$\mathrm{apply}(p, A) = A'$$

ossia, in forma compatta:

$$(r, m, A) \xrightarrow{\text{compile}} p \xrightarrow{\text{PatchApplied}} A'$$

Questa formula chiarisce che:

- la reaction rule sta al livello di *schema* o famiglia di trasformazioni;
- la patch sta al livello di *istanza concreta* e replayable della trasformazione;
- l'evento canonico registra la trasformazione concreta nella storia del trial.

Distinzioni da mantenere

Ne consegue che:

- la patch non coincide con il passo canonico in generale;
- la patch non coincide con Δ_V;
- `PatchApplied` è, nel prototipo attuale, il mutatore canonico della componente architetturale;
- `ReactionApplied` resta provenance accessoria o spiegazione trasformazionale.

Una singola trasformazione concreta può spesso essere descritta sia come `reaction rule + match` sia come `patch`, ma i due livelli non coincidono semanticamente.

A.7 Uso operativo di questa appendice

Nel resto della tesi questa appendice va letta come riferimento di stabilizzazione, non come capitolo autonomo di semantica. In particolare:

- il capitolo 5 resta il punto in cui replay, passo canonico, confronto rispetto a una vista e boundary vengono motivati semanticamente;
- questa appendice fissa il lessico minimo, la notazione e le distinzioni che i capitoli 5, 7 e 9 devono usare in modo coerente;
- i dettagli implementativi, serializzativi o infrastrutturali restano fuori da questo perimetro, salvo quando servono solo a chiarire il significato di un termine tecnico già introdotto.

A.8 Famiglie canoniche di cambiamento

Nel linguaggio della tesi, le famiglie canoniche di cambiamento di stato da richiamare quando serve sono:

- `create`
- `derive`
- `train`
- `proc`
- `obs`

Queste famiglie non sostituiscono la grammatica primaria, ma la organizzano in modo leggibile a livello di narrativa e analisi.

A.9 Tempo logico, tempo di processo e trace

Nel framework esistono almeno due nozioni di tempo:

- **tempo logico**: l'ordine dei record nel log, usato per replay deterministico;
- **tempo di processo**: epoche, step, budget e fase del processo, parte dello stato P.

Per un trial t con log eventi $\bar{e}^{(t)} = [e_1, \ldots, e_n]$ e stato iniziale $T_0^{(t)}$, scriveremo:

$$\text{replay}(T_0^{(t)}, \bar{e}^{(t)}) = \text{evolve}^*(T_0^{(t)}, \bar{e}^{(t)})$$

E indicheremo con $T_k^{(t)}$ lo stato dopo k eventi, cioè dopo k passi di tempo logico.

Termine	**Significato nel framework**
Esperimento	Contesto sperimentale condiviso che organizza un insieme di trial confrontabili. Nella notazione canonica è $E = \langle C, \mathcal{T} \rangle$.
Trial	Unità sperimentale primaria e tracciabile. Nella notazione canonica è $T = \langle A, H, W, P, O, L \rangle$; la sua storia è un log append-only di eventi.
Core del trial	Proiezione leggera del trial, usata quando serve parlare del suo nucleo senza portare con sé tutto il contesto sperimentale: $\text{core}(T) = \langle A, H, W \rangle$.
Modello (vista derivata)	Vista derivata del trial usata quando serve parlare del modello in senso stretto: $M(T) = \langle A, W \rangle$. Non è il trial completo.
Event sourcing	Strategia in cui la fonte di verità è una sequenza append-only di eventi; lo stato corrente si ottiene via replay deterministico. [8]
Replay	Ricostruzione deterministica dello stato applicando in sequenza gli eventi tramite `evolve`.
Command / Event	Un `Command` è una richiesta; `decide` la traduce in una lista finita di `Event`; solo gli eventi registrati mutano lo stato tramite `evolve`.
Boundary	Contratto verso strumenti esterni (training backend, export, import, validazione, strumenti bigrafici). Nel prototipo attuale è ispirato al paradigma delle *lenses* ed è replay-safe, ma non coincide con una formalizzazione completa di *lens with complements* come costrutto di primo livello. [7]
Vista	Proiezione read-only $V : \mathcal{T} \to Y_V$ che estrae dal trial una rappresentazione confrontabile.
Passo canonico (δ)	Nozione derivata che compatta un passo completo ottenuto da `decide` seguito da applicazione degli eventi tramite `evolve`. Non è il primitivo fondativo della semantica.
Confronto rispetto a una vista (Δ_V)	Differenza strutturata tra due trial sotto una vista scelta: $\Delta_V(T_1, T_2) := \text{diff}_V(V(T_1), V(T_2))$. È descrittivo/osservazionale, non un mutatore di stato.
Patch	Payload tecnico concreto di una mutazione architetturale; nel prototipo entra nella storia tramite `PatchApplied`.
Reaction rule	Schema generativo di riscrittura o spiegazione operazionale del cambiamento di A; nel prototipo `ReactionApplied` ha ruolo di provenance accessoria.
Lineage	Componente minimale del trial che ne fissa identità, parentela e sorgente immediata della derivazione: $L = \langle id, parent, src \rangle$.
Artifact store	Deposito per artefatti pesanti (checkpoint, report, export, visualizzazioni); nel log si memorizzano riferimenti stabili, non i file stessi.

Tabella A.1: Terminologia e concetti ricorrenti.

Bibliografia

[1] B. Archibald, M. Calder e M. Sevegnani, «Practical Modelling with Bigraphs,» 2024, Versione arXiv (31 Maggio 2024), 34 pagine. DOI: 10.48550/arXiv.2405.20745 arXiv: 2405.20745 [cs.LO]. indirizzo: https://arxiv.org/pdf/2405.20745.pdf

[2] L. Bartolomei, F. Tosi, M. Poggi e S. Mattoccia, «Stereo Anywhere: Robust Zero-Shot Deep Stereo Matching Even Where Either Stereo or Mono Fail,» in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025, pp. 1013–1027.

[3] J. Bergstra e Y. Bengio, «Random Search for Hyper-Parameter Optimization,» *Journal of Machine Learning Research*, vol. 13, n. 10, pp. 281–305, 2012. visitato il giorno 4 marzo 2026. indirizzo: https://www.jmlr.org/papers/v13/bergstra12a.html

[4] K. Claessen e J. Hughes, «QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs,» in *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, 2000, pp. 268–279. DOI: 10.1145/351240.351266 indirizzo: https://dl.acm.org/doi/10.1145/351240.351266

[5] T. Elsken, J. H. Metzen e F. Hutter, «Neural Architecture Search: A Survey,» *Journal of Machine Learning Research*, vol. 20, n. 55, pp. 1–21, 2019. indirizzo: https://www.jmlr.org/papers/v20/18-598.html

[6] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum e F. Hutter, «Efficient and Robust Automated Machine Learning,» in *Advances in Neural Information Processing Systems*, 2015. indirizzo: https://papers.nips.cc/paper/2015/hash/11d0e6287202fced83f79975ec59a3a6-Abstract.html

[7] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce e A. Schmitt, «Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem,» *ACM Transactions on Programming Languages and Systems*, vol. 29, n. 3, 2007. DOI: 10.1145/1232420.1232424 indirizzo: https://doi.org/10.1145/1232420.1232424

[8] M. Fowler, *Event Sourcing*, Online; accessed 2026-03-04, 2005. visitato il giorno 4 marzo 2026. indirizzo: https://martinfowler.com/eaaDev/EventSourcing.html

[9] FsCheck contributors, *FsCheck: Random Testing for .NET*, GitHub repository; accessed 2026-03-04, 2026. visitato il giorno 4 marzo 2026. indirizzo: https://github.com/fscheck/FsCheck

[10] X. He, K. Zhao e X. Chu, *AutoML: A Survey of the State-of-the-Art*, 2019. arXiv: 1908.00709 [cs.LG]. indirizzo: https://arxiv.org/abs/1908.00709

[11] F. Hutter, L. Kotthoff e J. Vanschoren, cur., *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2019. indirizzo: https://www.automl.org/wp-content/uploads/2019/05/AutoML_Book.pdf

[12] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh e A. Talwalkar, «Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization,» *Journal of Machine Learning Research*, vol. 18, n. 185, pp. 1–52, 2018. visitato il giorno 4 marzo 2026. indirizzo: https://www.jmlr.org/papers/v18/16-558.html

[13] R. Milner, «Bigraphical Reactive Systems,» in *CONCUR 2001 — Concurrency Theory*, ser. Lecture Notes in Computer Science, vol. 2154, Springer, 2001, pp. 16–35. DOI: 10.1007/3-540-44685-0_2

[14] R. Milner, *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009, ISBN: 978-0-521-73833-0. indirizzo: https://www.cl.cam.ac.uk/archive/rm135/Bigraphs-draft.pdf

[15] G. Perrone, S. Debois e T. T. Hildebrandt, «A Model Checker for Bigraphs,» in *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12)*, ACM, 2012, pp. 1320–1325. DOI: 10.1145/2245276.2231985

[16] J. Pineau et al., «Improving Reproducibility in Machine Learning Research (A Report from the NeurIPS 2019 Reproducibility Program),» *Journal of Machine Learning Research*, vol. 22, n. 164, pp. 1–20, 2021. visitato il giorno 4 marzo 2026. indirizzo: https://jmlr.org/papers/v22/20-303.html

[17] M. Poggi, A. Tonioni, F. Tosi, S. Mattoccia e L. Di Stefano, «Continual Adaptation for Deep Stereo,» *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, n. 9, pp. 4713–4729, 2022. DOI: 10.1109/TPAMI.2021.3126713

[18] «PROV-DM: The PROV Data Model,» World Wide Web Consortium (W3C), visitato il giorno 4 marzo 2026. indirizzo: https://www.w3.org/TR/prov-dm/

[19] P. Z. Ramirez et al., «Booster: A Benchmark for Depth From Images of Specular and Transparent Surfaces,» *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, n. 1, pp. 85–102, 2024. DOI: 10.1109/TPAMI.2023.3323858

[20] D. Sculley et al., «Hidden Technical Debt in Machine Learning Systems,» in *Advances in Neural Information Processing Systems 28 (NeurIPS 2015)*, 2015. DOI: 10.5555/2969442.2969519 indirizzo: https://papers.neurips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html

[21] M. Sevegnani e M. Calder, «BigraphER: Rewriting and Analysis Engine for Bigraphs,» in *Computer Aided Verification (CAV 2016)*, ser. Lecture Notes in Computer Science, vol. 9780, Springer, 2016, pp. 494–501. DOI: 10.1007/978-3-319-41540-6_27 indirizzo: https://eprints.gla.ac.uk/119384/13/119384.pdf

[22] J. Snoek, H. Larochelle e R. P. Adams, «Practical Bayesian Optimization of Machine Learning Algorithms,» in *Advances in Neural Information Processing Systems 25 (NeurIPS 2012)*, 2012. visitato il giorno 4 marzo 2026.

[23] C. Thornton, F. Hutter, H. H. Hoos e K. Leyton-Brown, «Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms,» in *Proceedings of KDD*, 2013. indirizzo: https://dl.acm.org/doi/10.1145/2487575.2487629

[24] A. Tonioni, F. Tosi, M. Poggi, S. Mattoccia e L. Di Stefano, «Real-Time Self-Adaptive Deep Stereo,» in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 195–204. DOI: 10.1109/CVPR.2019.00028

[25] TorchSharp contributors, *TorchSharp: A .NET library that provides access to LibTorch (PyTorch engine)*, GitHub repository; accessed 2026-03-04, 2026. visitato il giorno 4 marzo 2026. indirizzo: https://github.com/dotnet/TorchSharp

[26] F. Tosi, L. Bartolomei e M. Poggi, «A Survey on Deep Stereo Matching in the Twenties,» *International Journal of Computer Vision*, vol. 133, pp. 4245–4276, 2025. DOI: 10.1007/s11263-024-02331-0

[27] F. Tosi, A. Tonioni, D. De Gregorio e M. Poggi, «NeRF-Supervised Deep Stereo,» in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023, pp. 855–866.

[28] S. Wlaschin, *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*. The Pragmatic Bookshelf, 2018, ISBN: 9781680502541. indirizzo: https://pragprog.com/titles/swdddf/domain-modeling-made-functional/

[29] M. Zaharia et al., «Accelerating the Machine Learning Lifecycle with MLflow,» *IEEE Data Engineering Bulletin*, vol. 41, n. 4, pp. 39–45, dicembre 2018. visitato il giorno 4 marzo 2026. indirizzo: https://people.eecs.berkeley.edu/~matei/papers/2018/ieee_mlflow.pdf