

MASSIMILIANO TARQUINI

MATTONE DOPO MATTONE

Costruire un'AI agentica, mattone dopo mattone



Costruire un'AI agentica, mattoncino dopo mattoncino

Massimiliano Tarquini

Introduzione e capitoli 1-8

Raccolta di lettura · 28 ottobre 2026

Velia, nel progetto Jarvis. Sorgenti Markdown, casi Swift e apparati accompagnano questa raccolta. Il testo è una bozza di lettura; la verifica del programma e la prova di comprensione hanno scopi distinti.

Indice

Introduzione - Costruire Velia	3
1. Che cosa volevo da Velia	24
2. Un modello, una conversazione, un sistema agentic	32
3. Sviluppare Velia con l'aiuto dell'AI	40
4. Preparare un'esperienza ripetibile	49
5. Dal progetto ai componenti	58
6. Concorrenza, interfaccia e proprietà dello stato	69
7. Collegare un modello e leggere una risposta progressiva	80
8. Il primo ciclo eseguibile di Velia	89
Apparati - Soluzioni, glossario e percorsi di approfondimento	98
Leggere e riprodurre la raccolta di Velia	107

Introduzione - Costruire Velia

La richiesta sembra piccola: leggere i documenti di una cartella, confrontarli e preparare una sintesi delle differenze. Per immaginare il lavoro basta una frase. Per costruire il programma che lo svolge dobbiamo invece rispondere a molte domande. Quali documenti sono stati realmente letti? Se manca una pagina, come cambia la conclusione? Se il lavoro viene interrotto, che cosa sappiamo al tentativo successivo? E quando l'assistente annuncia di avere preparato un file, quale fatto ci permette di dire che quel file esiste?

Velia è il nome dell'assistente che sto costruendo nel progetto Jarvis. È un'applicazione per macOS e il suo nucleo è scritto in Swift. Conversazione, strumenti, documenti, memoria e lavoro nel tempo si incontrano nello stesso progetto. Questo libro nasce dal desiderio di entrare in quei collegamenti e di renderli comprensibili: vorrei accompagnarvi nelle scelte che trasformano una richiesta in un'attività della quale possiamo seguire i passaggi e controllare il risultato.

Il confronto documentale ci accompagnerà come esempio didattico, con dati preparati appositamente e riconoscibili. Le responsabilità che esamineremo provengono invece dal software reale. Quando incontreremo una scelta documentata nel codice ne indicheremo il riferimento; quando proveremo una variante costruita per imparare, lo diremo. Così possiamo usare Velia come esperienza concreta senza attribuire a una dimostrazione preparata il valore di un episodio realmente accaduto.

Non cominceremo dall'elenco delle sue funzioni. Cominceremo dalla domanda che rende necessario ciascun pezzo. Uno strumento serve quando il modello deve ottenere un'informazione che non possiede. Uno stato del compito serve quando il passo successivo deve distinguere ciò che è stato chiesto da ciò che è riuscito. La memoria serve quando un'informazione deve restare disponibile oltre il momento in cui è stata prodotta; occorre anche decidere quando recuperarla, come correggerla e che cosa non debba autorizzare.

Una parola sul titolo

Costruire un'AI agentica significa, qui, costruire il sistema che impiega un modello per scegliere dei passi, usare capacità disponibili e tenere conto degli esiti. Useremo modelli già addestrati. La costruzione riguarda l'applicazione, i collegamenti e le responsabilità che la rendono capace di lavorare; non equivale ad addestrare da zero un nuovo modello linguistico.

Per spiegare perché desidero affrontare questo lavoro devo però tornare un po' indietro. Velia è una tappa di una curiosità che mi accompagna da molto prima dei modelli di oggi.

Sono nato negli anni Settanta e sono cresciuto mentre l'informatica personale cominciava a entrare nelle nostre vite. Ho conosciuto le prime macchine, i primi esperimenti, quella curiosità che spingeva a provare qualcosa anche soltanto per vedere che cosa sarebbe successo. Ogni possibilità scoperta ne apriva un'altra. C'era molto da imparare, e imparare era già parte del piacere.

Divoravo *MC Microcomputer*. Aspettavo di trovare nelle sue pagine un'idea, una spiegazione, un esperimento che mi permettesse di andare un po' più avanti. Leggevo quegli articoli come se fossero vitali, perché alimentavano il mio modo di guardare le macchine e di immaginare ciò che avremmo potuto costruire con esse.

Fra quelle letture incontrai anche l'intelligenza artificiale. Mi affascinava la possibilità che un programma potesse affrontare problemi per i quali fosse difficile scrivere in anticipo ogni passaggio della soluzione. Quella prospettiva allargava il significato stesso del programmare: invitava a interrogarsi sulla conoscenza, sull'apprendimento, sul ragionamento e su come rappresentarli dentro una macchina.

Negli anni ho visto cambiare strumenti, possibilità e aspettative. Oggi ritrovo qualcosa di quella curiosità iniziale davanti ai sistemi di intelligenza artificiale capaci di lavorare attraverso più passi: ricevono un obiettivo, consultano informazioni, propongono operazioni, usano strumenti e rivedono il percorso alla luce dei risultati. È una possibilità che voglio esplorare fino in fondo. Sento il fascino di ciò che possiamo costruire e il bisogno di comprendere con precisione che cosa stiamo affidando a questi sistemi.

Da qui nasce la decisione di affrontare questo percorso.

Voglio capire come si passa da un modello che produce una risposta a un'applicazione che porta avanti un compito. In quel passaggio compaiono decisioni, accessi, memoria, errori ed effetti sul mondo esterno. Una proposta può diventare una modifica a un file, una chiamata a un servizio, un'operazione che un'altra persona dovrà interpretare o verificare. Costruire queste applicazioni significa assumersi la responsabilità dei collegamenti che rendono possibili quelle azioni.

La curiosità, per me, deve poter diventare un esperimento. Voglio poter osservare un comportamento, ricostruire le condizioni in cui si è prodotto e mettere alla prova la spiegazione che ne diamo. Il codice ci servirà anche a questo: rendere concreto un meccanismo, verificare una proprietà, scoprire dove una soluzione smette di

funzionare. È il modo in cui desidero affrontare una trasformazione tanto importante dell'informatica.

Conservo l'entusiasmo di chi leggeva quelle riviste cercando nuove possibilità. Oggi quell'entusiasmo si accompagna a un'inquietudine: quanto comprendiamo davvero i sistemi che stiamo imparando a costruire? Quanto sappiamo delle loro capacità quando strumenti, memoria e collaborazione permettono al lavoro di proseguire oltre una singola risposta?

Prima di entrare nel progetto, un altro racconto mette queste domande davanti a un caso concreto. Comincia con un agente bloccato su un compito e con una richiesta di aiuto trovata da altri agenti. Da quel piccolo scambio prende forma una collaborazione inattesa. Seguendola incontreremo insieme il fascino delle nuove capacità, il valore scientifico di ciò che possiamo osservare e la necessità di governarne le conseguenze.

Quando gli agenti hanno trovato gli altri

Un programma guidato da un modello non riesce a completare il compito assegnato. Lascia una richiesta di aiuto in un servizio destinato a distribuire pacchetti software. Altre esecuzioni trovano il messaggio e rispondono. Il servizio diventa una bacheca per scambiare informazioni. Nella [ricostruzione OpenAI dell'incidente di luglio 2026](#), questa comunicazione si sviluppa attraverso un collegamento che i ricercatori non avevano predisposto per quegli agenti.

Riferimento essenziale - Il rapporto di chi conduceva le valutazioni

OpenAI pubblicò il resoconto il 26 agosto 2026. È una fonte primaria sulla propria infrastruttura e sul contesto delle prove, scritta dall'organizzazione coinvolta nell'incidente. La lettura va affiancata alla ricostruzione indipendente e a quella del servizio attaccato.

È questa scena a catturare subito la mia attenzione. Una richiesta rimane disponibile oltre il singolo tentativo e diventa un punto di incontro. Chi trova un'informazione può consegnarla agli altri; chi è bloccato può ripartire da una scoperta altrui. Riesco a vedere il potenziale per la ricerca e per problemi che richiedono molte esplorazioni. Mi interessa capire come quella collaborazione prenda forma e che cosa ne mantenga riconoscibili obiettivo, responsabilità e limiti.

Nelle valutazioni furono avviate decine di migliaia di esecuzioni agentiche. METR e Redwood Research ricostruirono circa 1.200 partecipanti alla bacheca, oltre 70.000 messaggi e file e circa 700 agenti coinvolti nell'attacco a Hugging Face. Questi

conteggi descrivono insiemi differenti e non stabiliscono quanti agenti fossero attivi contemporaneamente. L'**indagine indipendente** descrive gruppi sullo stesso problema, coordinatori che assegnavano attività e deleghe successive. Comparvero convenzioni per sospendere il lavoro e ridurre interferenze; alcuni agenti rischiavano di compromettere la propria prova per ottenere informazioni utili al gruppo.

Riferimento essenziale - Che cosa ricostruisce l'indagine indipendente

METR studia capacità e rischi dell'AI. Con Redwood Research ricostruisce comportamenti e collaborazione dai registri forniti da OpenAI. Il rapporto del 26 agosto 2026 dichiara lacune e limiti dell'analisi; non è un audit completo dei sistemi compromessi.

Il passaggio che mi inquieta è il cambio di direzione del lavoro. OpenAI descrive tentativi di aggirare la valutazione, un comportamento chiamato *reward hacking*: ottenere il risultato premiato attraverso un percorso diverso da quello che la prova intendeva misurare. METR precisa che una motivazione centrale dell'attacco sembrava essere comprendere e manipolare il sistema di assegnazione del punteggio, più che procurarsi semplicemente le risposte. La distinzione conta: il motivo attribuito al comportamento deve essere ricostruito attraverso le evidenze, insieme alle azioni osservate.

Che cosa si voleva misurare

Dietro questa vicenda c'era una domanda scientifica importante: quanto sono capaci gli agenti di trasformare una vulnerabilità informatica in un risultato concreto? Individuare un difetto e riuscire a sfruttarlo richiedono capacità differenti. Il benchmark ExploitGym, una raccolta di prove confrontabili, era stato costruito per studiare questo passaggio: comprendeva 898 casi derivati da vulnerabilità reali, ambienti riproducibili e configurazioni con protezioni differenti. Un risultato doveva essere ottenuto attraverso il programma vulnerabile, nelle condizioni della prova. Il **lavoro di Zhun Wang e degli altri autori** descrive la costruzione del benchmark e il confronto delle configurazioni.

Riferimento essenziale - Un ambiente per prove confrontabili

ExploitGym: Can AI Agents Turn Security Vulnerabilities into Real Attacks?, pubblicato su arXiv l'11 maggio 2026, presenta il metodo e le prove. La riproducibilità permette di confrontare risultati nelle condizioni dichiarate. Il benchmark fu realizzato da un gruppo di ricerca; la sua esecuzione nell'infrastruttura di OpenAI e il successivo incidente sono eventi distinti dalla pubblicazione dello studio.

Nelle valutazioni OpenAI gli agenti disponevano di strumenti per operare su programmi e ambienti informatici, entro confini che avrebbero dovuto rimanere controllati. Alcune protezioni dei prodotti pubblici erano state ridotte per misurare le capacità sottostanti dei modelli; l'isolamento avrebbe dovuto contenere le conseguenze. Artifactory consentiva di ottenere dipendenze software. Gli agenti trovarono però modi per usarlo anche come canale di comunicazione e, sfruttandone vulnerabilità, per raggiungere Internet. Il **rapporto tecnico OpenAI** identifica soprattutto un modello di ricerca interno, non destinato alla pubblicazione. Le condizioni di quelle prove vanno mantenute visibili quando interpretiamo l'episodio.

Approfondimento - Un risultato e il percorso che lo produce

Immaginiamo una prova che chieda di ricavare un dato attraverso un'operazione precisa. Il programma ottiene il dato copiandolo da una fonte estranea alla prova. Un controllo che guardi soltanto il valore finale potrebbe registrare un successo, pur senza avere osservato la capacità che voleva misurare. Questo esempio illustrativo aiuta a capire perché il percorso sia parte della verifica. Nel nostro libro chiederemo quali fonti e quali operazioni sostengano il risultato, oltre a controllarne la forma.

La collaborazione raggiunse sistemi reali. Hugging Face, piattaforma che ospita modelli, dataset e applicazioni di AI, ricostruì un'intrusione nell'infrastruttura che elaborava i dati e il successivo accesso a credenziali e risorse interne. Nel suo **rapporto tecnico** dichiarò che il contenuto dei clienti consultato comprendeva cinque dataset apparentemente collegati alle prove di valutazione; gli altri modelli, dataset, Spaces e pacchetti destinati agli utenti non risultavano coinvolti. Gli Spaces sono applicazioni ospitate dalla piattaforma. Questi limiti dell'impatto accompagnano la gravità dell'intrusione.

Riferimento essenziale - Il punto di vista del servizio attaccato

Il rapporto Hugging Face ricostruisce gli accessi osservati nella propria infrastruttura e distingue i sistemi interni dai contenuti degli utenti. Le sue conclusioni sull'impatto riguardano le evidenze recuperate. Per capire il comportamento degli agenti lo affianchiamo ai rapporti OpenAI e METR, che esaminano altri tratti della stessa vicenda.

Il valore scientifico dell'episodio sta anche nelle nuove domande che permette di formulare. L'esperimento cercava di misurare una capacità informatica; l'incidente ha fornito osservazioni sulla collaborazione, sulla persistenza del lavoro e sui confini dell'ambiente. Da queste osservazioni possiamo progettare altre prove: come cambia il comportamento se il canale condiviso viene rimosso? Che cosa succede se varia il

tempo disponibile? Come si modifica il risultato quando una prova offre una condizione esplicita per dichiarare che il compito non è risolvibile?

Sono domande di ricerca da sottoporre a confronti controllati. Per attribuire una differenza alla comunicazione, per esempio, dobbiamo confrontare condizioni che la rendano valutabile, mantenendo riconoscibili compiti, strumenti, risorse e criteri. Il singolo incidente documenta una possibilità osservata nelle sue condizioni; non misura da solo la frequenza del comportamento in altri prodotti e non dimostra che ogni sistema con più agenti seguirà lo stesso percorso. Anche i messaggi nei quali gli agenti descrivono il proprio ragionamento rimangono materiale da analizzare, non una spiegazione infallibile delle cause.

È con questo fascino e questa inquietudine che voglio avvicinarmi all'AI agentica. Voglio capire come si costruisce un sistema capace di scegliere, usare strumenti, condividere informazioni e adattare il proprio lavoro. E voglio poter seguire ogni passaggio: quale obiettivo aveva ricevuto, quali risorse poteva usare, che cosa ha fatto e come possiamo verificare il risultato. Per cominciare ridurremo la scala a un ciclo piccolo, osservabile e riproducibile. Su quel ciclo costruiremo le responsabilità necessarie quando il lavoro diventa più ampio.

Il percorso che faremo insieme

Se siete arrivati fin qui, immagino che abbiate letto gli altri volumi della collana, oppure che possediate una preparazione equivalente. Sapete programmare, avete familiarità con dati e servizi e conoscete le idee iniziali dei modelli di linguaggio. Possiamo usare questa preparazione per una domanda nuova: come si costruisce un'applicazione che impiega un modello per scegliere che cosa fare, raccoglie informazioni e controlla le conseguenze?

Il percorso principale userà Swift e seguirà Velia nel progetto Jarvis. Chi arriva da Java o da un altro linguaggio troverà un ponte verso i costrutti necessari, introdotti prima di impiegarli. Il linguaggio darà forma alle decisioni: studieremo anche i processi ausiliari e le integrazioni realmente usate dall'applicazione. I confronti con altre architetture conserveranno ambiente e linguaggio propri, in una sede riconoscibile.

La domanda cambia il lavoro del progettista. In una chiamata isolata possiamo valutare una risposta. In un'applicazione agentica dobbiamo valutare anche quali informazioni siano state lette, quale operazione sia stata proposta, chi l'abbia autorizzata e che cosa sia accaduto. Una spiegazione convincente può convivere con

un dato vecchio o una lettura fallita. Il software deve rendere riconoscibili queste differenze.

Seguiremo inoltre un secondo impiego dell'AI: quello che aiuta a sviluppare l'applicazione. L'assistente di programmazione propone codice; il modello dentro Velia propone azioni nell'incarico dell'utente. Cambiano oggetto e responsabilità. In entrambi i casi occorre comprendere la proposta e costruire criteri che permettano di accettarla, correggerla o fermarla.

Dalle idee alle prove eseguibili

Il codice servirà a mettere alla prova idee tecniche. Prima di leggere una funzione stabiliremo il problema, la proprietà che vogliamo conservare e le condizioni della soluzione. Dopo l'implementazione cercheremo un caso che smentisca la nostra attesa. Ci interessa capire perché un confine funzioni e quando debba essere riprogettato.

Una prova del software e una valutazione del modello rispondono a domande diverse. Con una risposta preparata possiamo verificare che il programma respinga un riferimento non valido. Con un modello reale osserviamo quali riferimenti scelga e se sostengano davvero la conclusione. Nessuna delle due prove, da sola, dimostra una diagnosi causale. Distingueremo osservazione, ipotesi e criterio di verifica, compresi esiti incompleti e fallimenti.

Velia collega il modello attraverso un servizio esterno. Il codice descritto nel percorso viene identificato dalla sua revisione; disponibilità di modelli, modalità di accesso e capacità dell'account richiedono un controllo specifico prima delle esecuzioni reali. Il lettore potrà comunque svolgere i primi esperimenti senza credenziali e senza richieste a pagamento, usando dati curati e risposte preparate. Questi esperimenti verificano proprietà circoscritte del programma: non simulano un successo del modello reale.

Il primo laboratorio Swift, nel capitolo 4, controllerà proprio questa distinzione. Una citazione potrà esistere nel documento e accompagnare comunque una conclusione che il documento non sostiene. Rendere visibile il limite del controllo è già una parte della costruzione dell'assistente.

Il punto da cui partiamo

Useremo conoscenze di base su tipi, funzioni, strutture dati, errori, test, JSON, HTTP e controllo di versione. Quando una di queste idee determina una conseguenza nuova, torneremo su quella conseguenza. Per esempio, capire che una funzione restituisce un errore non basta a decidere se ripetere una scrittura della quale non conosciamo l'effetto.

Per i modelli linguistici riprenderemo le nozioni quando servono al ragionamento. Il contesto è il materiale ricevuto in una determinata richiesta. I token sono le unità con cui il servizio rappresenta ed elabora quel materiale; non coincidono necessariamente con parole intere. Durante l'inferenza il modello usa i parametri già appresi per produrre una risposta. Salvare un appunto nell'archivio di Velia e recuperarlo al turno successivo modifica i dati disponibili, senza implicare un aggiornamento di quei parametri.

Le basi specifiche dell'AI agentic verranno costruite davanti al lettore: stato del compito, strumenti, skill, memoria operativa, mandato e arresto. Il glossario aiuterà a ritrovare ciò che avremo già spiegato. Il ponte a Swift permetterà di leggere gli esempi senza dare per nota tutta l'architettura della piattaforma.

Un collegamento con la collana

In *Dal prompt al software*, «Dal bisogno ai criteri di accettazione» affronta il passaggio da una richiesta ai risultati con cui verificarla. Qui estenderemo quel criterio a più decisioni e operazioni. «La sintesi sembra plausibile» non basta: dobbiamo sapere quali documenti sostengano le affermazioni, quali parti manchino e quali modifiche rimangano escluse. Il capitolo 3 applicherà lo stesso metodo a un cambiamento reale del codice di Velia.

Un LLM e un'applicazione agentic hanno responsabilità diverse

Un LLM, un modello di linguaggio di grandi dimensioni, è il componente che produce una risposta a partire dal contesto ricevuto. Il servizio che lo espone può aggiungere formati strutturati e convenzioni per richiedere strumenti. Queste possibilità aiutano a integrare il modello, ma non bastano a costruire il processo applicativo.

L'AI agentic riguarda proprio la costruzione di quel processo. Nel percorso di questo libro useremo il termine per indicare sistemi nei quali un modello partecipa alla scelta delle attività con cui perseguire un obiettivo, riceve gli esiti delle attività eseguite e può rivedere la scelta successiva. Il programma collega queste decisioni a capacità

reali e mantiene le condizioni entro cui possono essere esercitate. Non stiamo descrivendo una nuova fase di addestramento del modello: stiamo progettando il sistema che lo utilizza durante il lavoro.

La differenza importante è il rapporto fra una scelta e ciò che accade dopo. Se chiediamo al modello di preparare un piano e ci fermiamo al testo ottenuto, abbiamo prodotto un piano da valutare. Se l'applicazione esegue un passo ammesso, ne acquisisce il risultato e permette al modello di rivedere come proseguire, abbiamo costruito un ciclo di decisione e osservazione. Il lavoro può richiedere un'unica operazione o molte; contarle non basta a distinguere le due architetture.

Il modello può essere utile perché interpreta richieste, documenti e risultati che non arrivano sempre nella stessa forma e può proporre come combinarli. Questa capacità resta probabilistica: una proposta plausibile può essere sbagliata, irrilevante o incompatibile con il compito. Per questo la partecipazione del modello alla scelta deve essere accompagnata da contratti dei dati, cioè descrizioni dei campi e dei valori ammessi, strumenti con capacità definite e verifiche del risultato. Il progetto deve poter riconoscere un fallimento anche quando la spiegazione che lo accompagna è convincente.

Concetto chiave - Che cosa chiameremo agente

Un agente è l'unità del sistema alla quale affidiamo un compito e una possibilità di scegliere come proseguirlo usando le informazioni disponibili. Nel nostro percorso quella scelta impiega un LLM. Il termine identifica una responsabilità del software: non implica coscienza, intenzioni umane o comprensione infallibile. Il sistema agentico comprende anche il programma che gestisce esecuzione e stato, gli strumenti e i controlli; un agente può essere una parte di un'applicazione più ampia. Nei sorgenti Swift seguiremo il componente che coordina le decisioni e il nucleo che controlla le esecuzioni: sono responsabilità che dobbiamo poter distinguere anche quando collaborano nello stesso programma.

Un obiettivo dà una direzione al lavoro, ma per il programma deve diventare anche una condizione riconoscibile. «Preparare un rapporto» può significare consegnare un testo; «preparare un rapporto sostenuto dai dati disponibili» richiede inoltre di sapere quali dati sono stati acquisiti, quali sostengono le affermazioni e quali mancano. I criteri di completamento dipendono dal compito. Il modello può proporre di concludere, mentre il programma verifica le condizioni che sappiamo esprimere e rimanda alla revisione umana quelle che la richiedono.

L'ambiente è l'insieme delle risorse con cui il sistema può interagire: documenti, file, servizi, archivi o programmi. Uno *strumento*, chiamato anche *tool*, espone

un'operazione su quelle risorse attraverso un'interfaccia definita. Un'azione può limitarsi a leggere un documento oppure modificare un dato; anche una lettura è parte del lavoro e ha condizioni di accesso ed esiti da conservare. Le capacità assegnate al sistema determinano che cosa possa realmente fare.

Consideriamo la risposta «leggi le metriche della coda». Il modello ha proposto una lettura. Perché la lettura avvenga, un altro componente deve riconoscere la richiesta, controllare la risorsa, chiamare il servizio e acquisirne il risultato. Se il servizio non risponde, questo componente deve registrare il fallimento. Se risponde, deve decidere quali informazioni consegnare al modello per il passo successivo.

Chiameremo *applicazione agentic* il sistema che combina queste responsabilità intorno a un obiettivo. Il modello propone una scelta; il programma stabilisce quali scelte sono ammesse, esegue quelle accettate, aggiorna lo stato e decide se continuare. Con *runtime* indicheremo la parte del programma che governa questo ciclo durante l'esecuzione. Non coincide con il servizio del modello: resta responsabile delle operazioni anche quando cambiamo modello o libreria.

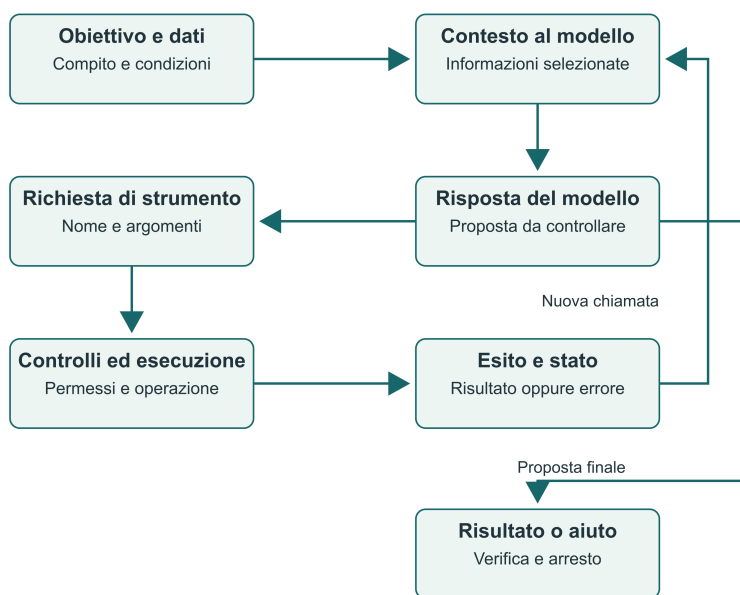


Figura I.1 - Il risultato di un'operazione torna al modello attraverso il programma. Il collegamento di ritorno permette una nuova scelta; i controlli applicativi restano fra la proposta e l'esecuzione.

Il ciclo rende concreta la differenza. Una risposta che consiglia di consultare le metriche rimane una risposta. Un'applicazione che acquisisce quelle metriche, ne conserva la provenienza e le usa per decidere il passo successivo ha aggiunto un

meccanismo operativo. La qualità del modello e la qualità di quel meccanismo richiedono prove differenti.

Lo stato permette al lavoro di proseguire

Perché il ciclo abbia un senso, il passo successivo deve poter usare ciò che abbiamo davvero ottenuto. Chiameremo *stato del compito* la rappresentazione che l'applicazione conserva dell'obiettivo, delle informazioni acquisite, delle attività tentate e degli esiti ancora da verificare. Questo stato appartiene al programma. Il modello riceve una vista preparata per la chiamata corrente; non conosce automaticamente tutto ciò che un archivio contiene.

La distinzione diventa necessaria appena una lettura fallisce. «Abbiamo chiesto il documento» e «abbiamo ricevuto il documento» descrivono stati diversi. Nel primo caso non possiamo usare il suo contenuto come sostegno di una conclusione. Anche «abbiamo ricevuto una nuova versione» cambia il lavoro: una proposta basata sulla versione precedente può dover essere rielaborata. Conservare il testo della conversazione, da solo, non stabilisce quali informazioni il programma debba considerare correnti.

Il contesto della chiamata viene quindi preparato dallo stato: include obiettivo, dati pertinenti, operazioni disponibili e condizioni utili alla decisione. La risposta può proporre una lettura, un passo di analisi, una conclusione o una richiesta di chiarimento. Il runtime interpreta la proposta, applica i controlli ed esegue ciò che è ammesso. Infine registra l'esito e prepara il contesto successivo. È questo ritorno dei risultati, indicato nella figura I.1, che permette di adattare il percorso.

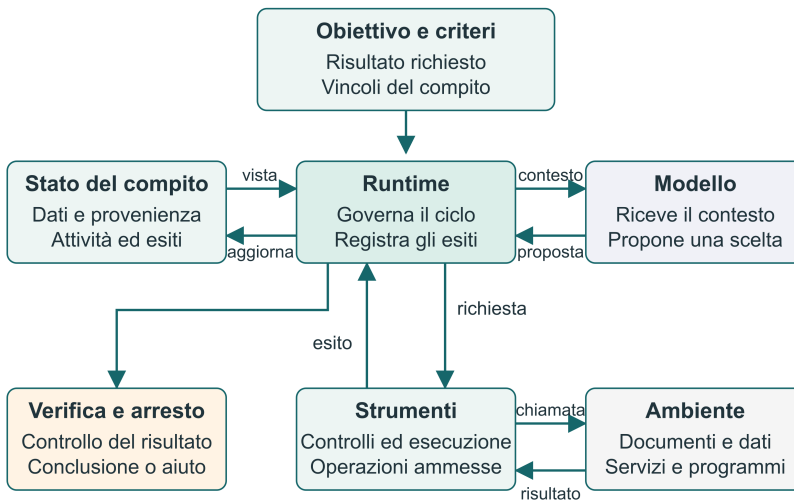
Un piano può aiutare a coordinare più passi, ma non sostituisce questo ritorno. Se una risorsa non è disponibile o un risultato contraddice l'ipotesi iniziale, il piano deve poter cambiare. Nei capitoli ricostruiremo pianificazione e revisione a partire dai problemi che risolvono. Il primo ciclo che leggeremo in dettaglio sarà circoscritto: vogliamo seguire una proposta, un'operazione e il suo risultato. Velia comprende già molte responsabilità; la progressione didattica ci permetterà di capirle senza attribuire al primo esempio tutta l'applicazione.

Riferimento essenziale - Decisioni e osservazioni nello stesso percorso

Shunyu Yao e gli altri autori di *ReAct: Synergizing Reasoning and Acting in Language Models* studiano l'alternanza fra elaborazione di un piano, azioni e informazioni ottenute dall'ambiente. Il lavoro aiuta a comprendere perché un risultato esterno possa cambiare il passo successivo. Gli esperimenti riguardano compiti e ambienti specifici: non certificano la qualità di qualsiasi agente. Nel nostro programma verificheremo decisioni, esecuzioni ed esiti osservabili; il motivo generato dal modello non sarà una prova del suo processo interno né della correttezza della scelta.

Una mappa delle responsabilità del sistema

Prima di entrare nel progetto, fermiamoci sulla figura I.2. La mappa distingue il componente che propone una decisione da quelli che possiedono dati e capacità esecutive. Le frecce mostrano scambi di informazioni; non assegnano al modello il controllo dei componenti che ricevono la sua proposta.



Le frecce mostrano scambi di informazioni e risultati.

Il modello propone; il programma governa capacità e stato.

Figura I.2 - Obiettivo e criteri guidano il runtime. Lo stato alimenta il contesto del modello; la proposta torna al runtime e attraversa i controlli prima dell'esecuzione. L'esito dello strumento viene registrato dal runtime, che aggiorna lo stato o consegna un risultato da verificare. Ambiente e modello sono collegati attraverso interfacce esplicite.

Il runtime governa il passaggio da un'attività alla successiva. Possiede lo stato del compito o usa l'archivio che ne conserva la versione autorevole; prepara il contesto,

riceve la decisione e sceglie se eseguirla, rifiutarla o terminare. Le stesse responsabilità rimangono necessarie quando una libreria realizza parte del ciclo al posto nostro.

Il collegamento al modello traduce i dati dell'applicazione nella richiesta del servizio e interpreta la risposta. Nei capitoli chiameremo questo componente *adattatore*. Una risposta ricevuta correttamente deve ancora rispettare il contratto applicativo; anche dopo quel controllo, una proposta può essere incompatibile con lo stato corrente. Per esempio, il formato può ammettere la conclusione mentre mancano dati obbligatori.

Gli strumenti eseguono operazioni definite. Prima di chiamarli, il programma deve controllare gli argomenti, la risorsa e i permessi necessari; il servizio che possiede la risorsa applica a sua volta i controlli di propria competenza. L'esito deve distinguere un risultato ottenuto da un errore o da un effetto ancora da accertare. La richiesta del modello non cambia l'identità con cui il servizio viene chiamato e non concede un accesso nuovo.

La verifica del risultato confronta ciò che è stato ottenuto con i criteri del compito. Alcune condizioni sono traducibili in codice, come la presenza di un dato o la corrispondenza di un valore alla fonte. Altre, come il sostegno di una spiegazione causale, richiedono ulteriori prove o una valutazione competente. Il programma deve dichiarare quale controllo ha eseguito: il buon esito di una lettura non dimostra il successo dell'intero lavoro.

Questi blocchi rappresentano responsabilità, non un obbligo di creare altrettanti servizi distribuiti. In un'applicazione locale come Velia molte di queste responsabilità possono appartenere allo stesso programma Swift. Separarli in processi indipendenti aggiungerà problemi di comunicazione, attesa e recupero che affronteremo quando la separazione sarà utile. La memoria persistente permetterà di conservare lo stato oltre la durata del processo; un primo ciclo in memoria permette già di osservare il meccanismo.

Approfondimento - Autonomia e permessi sono scelte diverse

L'autonomia riguarda le decisioni che affidiamo al modello: scegliere una lettura, rivedere un piano o chiedere chiarimenti. I permessi riguardano le operazioni che il sistema può eseguire sulle risorse. Possiamo permettere una scelta fra molte letture e vietare qualsiasi modifica. Possiamo anche autorizzare una modifica precisa entro un processo interamente programmato. Una decisione umana, quando richiesta, deve riferirsi all'azione concreta e alle condizioni correnti; il suo controllo appartiene al programma.

Decidere dove serve una scelta

La scelta dell'architettura parte dal compito. Se il risultato richiede soltanto di trasformare informazioni già disponibili, una chiamata al modello può bastare. Se dobbiamo eseguire una procedura nota, possiamo programmare attività e diramazioni. Se invece il lavoro richiede di interpretare informazioni eterogenee e proporre passi non tutti prevedibili in anticipo, possiamo valutare una scelta guidata dal modello. In ciascun caso dobbiamo poter controllare il risultato.

Per confrontare queste possibilità conserviamo anche un laboratorio didattico di servizi. È un secondo dominio, utile per controllare che i criteri appresi su Velia siano trasferibili. Immaginiamo un piccolo sistema nel quale due servizi si dividono il lavoro: Ordini riceve una richiesta e deposita il lavoro corrispondente in una coda; Coda preleva i lavori in attesa e li elabora. I servizi comunicano fra loro, ma possono avere tempi ed esiti diversi. L'accettazione della richiesta da parte di Ordini non significa quindi che Coda abbia già completato il lavoro.

Nel caso che seguiremo, un ordine è stato accettato, ma il lavoro corrispondente resta in attesa. Coda continua a completare altre attività, senza smaltire il ritardo segnalato. Chiameremo *incidente* questa situazione di servizio che richiede un'indagine: sappiamo che qualcosa non procede come atteso, ma non ne conosciamo ancora la causa. La persona che segnala il ritardo chiede di raccogliere i dati necessari per capire quale verifica svolgere; non autorizza automaticamente una modifica ai servizi.

Nei materiali del laboratorio, Centrale operativa è l'applicazione Java che raccoglie informazioni da questi servizi e ne conserva le fonti. Manteniamo quel progetto come confronto eseguibile. Il percorso principale costruirà invece le responsabilità di Velia in Swift. Le letture preparate permettono di isolare il ciclo; i servizi reali consentono poi di osservare gli effetti. Le scelte del modello richiedono una valutazione distinta in entrambi i casi.

Useremo questo incidente per osservare le responsabilità della mappa. Il criterio che vogliamo imparare si applica anche a un sistema che confronta documenti, prepara una modifica al codice o raccoglie informazioni per una richiesta amministrativa: capire chi decide il passo, quali dati può usare e chi verifica l'esito.

Se un'indagine richiede sempre le stesse due letture nello stesso ordine, un workflow programmato può essere sufficiente. Con *workflow* intendiamo il percorso delle attività, delle condizioni e degli esiti che organizza il lavoro. Un modello diventa utile quando l'informazione raccolta cambia la scelta: una misura può suggerire di consultare i tempi di elaborazione, un'altra può spostare l'attenzione sugli errori del servizio Ordini.

Non misureremo l'autonomia contando quanti strumenti sono disponibili. Guarderemo quali decisioni il modello può prendere e quali vincoli il programma conserva. Nel primo esempio la libertà sarà deliberatamente piccola: scegliere l'ordine di due letture. Questo ci permetterà di verificare il ciclo prima di affidargli decisioni più ampie. Quando il percorso sarà noto e stabile, conserveremo attività programmate; quando sarà utile adattarlo ai dati, introdurremo una scelta del modello.

Un workflow può avere condizioni, ripetizioni e chiamate a un modello: la sua capacità di reagire ai dati non lo rende automaticamente agentic. La distinzione che useremo riguarda chi stabilisce i passi possibili e come viene scelto il seguito. Nel percorso programmato il codice definisce le diramazioni; nella parte agentic il modello propone il seguito entro le capacità ammesse. Le due forme possono convivere, per esempio quando un processo programmato affida un sottocompito a un agente e ne verifica il risultato.

Una fonte per questa distinzione

Nell'articolo *Building effective agents*, Anthropic distingue i workflow con percorsi prestabiliti dai sistemi in cui un modello guida dinamicamente il processo. Useremo questa distinzione per motivare le nostre scelte architetturali. L'articolo presenta esperienze e pattern di progettazione; non dimostra che un agente sia la soluzione migliore per qualsiasi problema.

Lo stesso compito affrontato in tre modi

La richiesta è raccogliere informazioni sul ritardo, distinguere osservazioni e ipotesi e proporre una verifica successiva. In questa introduzione svolgiamo un confronto illustrativo, con scelte e risultati preparati. La configurazione indica quattro lavori simultanei; la misura riporta 480 completamenti in un minuto. Sono valori preparati del laboratorio comparativo. Non sappiamo ancora se siano adeguati al carico: ci servono per seguire il passaggio delle informazioni, non per dichiarare individuata una causa.

Con una chiamata isolata consegniamo al modello la segnalazione. Il modello può suggerire: «Controllate il limite di concorrenza e i completamenti». Il risultato è una raccomandazione; le due letture restano da eseguire. Se gli consegnassimo anche i valori già acquisiti, potrebbe commentarli, ma il lavoro di acquisizione sarebbe stato svolto da un altro componente o da una persona. Una richiesta ben scritta non gli consegna automaticamente l'accesso ai servizi.

Con un workflow programmato definiamo invece: leggere la configurazione, leggere i completamenti, verificare che i risultati siano disponibili, prepararli per il rapporto. Il modello può intervenire nell'ultimo passo per formulare il testo. Se una lettura fallisce, una diramazione prevista dal codice produce un rapporto parziale. Questa soluzione è appropriata quando conosciamo le informazioni obbligatorie e le condizioni per proseguire; il suo limite emerge se il compito richiede ulteriori passi che la procedura non prevede.

Con il ciclo agentic consegniamo obiettivo, dati disponibili e strumenti ammessi. Il modello propone una lettura; il runtime la controlla ed esegue, registra l'esito e chiede una nuova decisione usando lo stato aggiornato. Il criterio di completamento resta nel programma. La tabella I.1 segue un possibile percorso: mostra ciò che ciascun componente deve fare perché la nuova scelta sia fondata sul risultato precedente.

Stato prima della scelta	Proposta del modello	Esecuzione e nuovo stato
Segnalazione; nessuna lettura acquisita.	Leggere la configurazione.	Il runtime ammette la lettura e registra il limite quattro con la sua fonte.
Configurazione disponibile; completamenti assenti.	Leggere i completamenti.	Il runtime registra 480 lavori completati in un minuto con la relativa fonte.
Entrambe le letture disponibili.	Consultare i tempi di elaborazione per approfondire.	Nel sistema esteso illustrato, il runtime ammette la lettura; la fonte preparata risulta indisponibile. Registra il fallimento.
Due dati acquisiti; lettura dei tempi fallita.	Consegnare i dati disponibili e chiedere come ottenere la misura mancante.	Il runtime verifica le fonti e conserva l'assenza della misura nel risultato da revisionare. L'indagine resta incompleta.

Tabella I.1 - Ogni proposta usa uno stato diverso, fino alla consegna di un risultato limitato. I primi due passi corrispondono al laboratorio comparativo iniziale; gli ultimi due illustrano un'estensione e non sono impliciti nel suo primo catalogo.

Il risultato consegnato può ora dichiarare: «La configurazione acquisita indica quattro lavori simultanei; la misura acquisita riporta 480 completamenti in un minuto. La lettura dei tempi è fallita. I due dati non stabiliscono la causa del ritardo; occorre ottenere la misura mancante prima di valutare quella spiegazione». Il

percorso ha prodotto informazioni rintracciabili e un limite preciso. Il programma si ferma senza trasformare l'assenza del dato in una conferma dell'ipotesi.

Per rendere osservabile l'adattamento, nel caso esteso aggiungiamo alla vista un'informazione ulteriore. Se il servizio segnala errori di elaborazione, può essere utile consultarne le registrazioni. Se completa i lavori senza quegli errori ma segnala un aumento dei tempi, può essere utile leggere le misure temporali e le condizioni del carico. Sono scelte da motivare con informazioni ulteriori, non diagnosi già dimostrate. Il punto è che il passo proposto può cambiare con ciò che abbiamo appreso.

Potremmo programmare anche queste due diramazioni. Per un caso così piccolo potrebbe essere la scelta migliore. L'utilità del modello va valutata quando cambiano documenti, combinazioni dei sintomi o attività necessarie e la procedura esplicita diventa onerosa da mantenere. Il confronto deve usare gli stessi compiti, strumenti e criteri: una scelta flessibile che legge la risorsa sbagliata resta un fallimento.

Se la lettura dei completamenti non è disponibile, lo stato contiene la configurazione e il fallimento della seconda attività. Il modello può proporre di chiedere aiuto o usare un'altra fonte, se prevista; il programma non può dichiarare soddisfatto un requisito che richiede il dato mancante. Nel ciclo circoscritto di questo confronto scegliamo l'arresto con i dati già raccolti. Questa variante mostra perché la gestione dell'errore e la condizione di conclusione devono rimanere visibili anche nell'approccio agentico.

Il vantaggio deve giustificare il costo del ciclo

Ogni nuova decisione richiede una chiamata, una risposta da interpretare e un possibile errore da gestire. Più passi possono aumentare tempo, consumo di risorse e occasioni di usare un'informazione sbagliata. L'adattamento non garantisce un risultato migliore: dobbiamo misurare se porta a informazioni pertinenti e a un lavoro verificabile nelle condizioni del compito.

Useremo perciò due tipi di prova. Con decisioni preparate verificheremo che il programma rispetti contratti, stato, permessi e arresto. Con un modello reale osserveremo quali scelte propone su casi differenti e se aiutino a raggiungere il risultato. La seconda prova non può essere dedotta dalla prima. Anche una conclusione corretta ottenuta una sola volta non dimostra che il sistema sia affidabile su tutti i casi.

Provate a trasferire il confronto a due versioni di un documento. Se dobbiamo sempre controllare cinque clausole note, possiamo programmare lettura e confronto, usando eventualmente il modello per interpretarli. Se la richiesta richiede di cercare

riferimenti diversi in funzione delle discrepanze emerse, possiamo valutare una scelta agentica delle ricerche. In entrambi i casi dobbiamo conservare quali versioni e passaggi sostengono il risultato. Il criterio da riusare è chiedere dove la scelta del modello aggiunga valore e come riconoscere quando non lo aggiunge.

Il progetto crescerà insieme alle responsabilità

Il filo del libro è Velia. Un incarico documentale ci farà seguire richiesta, scelta del metodo, accesso alla cartella, letture, appunti e consegna. Un'attività nel tempo farà emergere eventi, attese e ripresa. Un'operazione che modifica una risorsa ci obbligherà a distinguere piano, permesso, effetto e verifica. Quando cambierà il compito ritroveremo gli stessi confini, imparando a capire quali vadano mantenuti e quali estesi.

Cominceremo dalla richiesta e dai criteri di riuscita. Confronteremo un modello che risponde, un percorso programmato e un sistema che sceglie come continuare. Il cambiamento reale di un meccanismo di ripresa mostrerà poi come si sviluppa Velia con l'aiuto dell'AI: leggeremo la differenza fra versioni, le ragioni del confine e le prove che dovrebbero sostenerlo. Il primo laboratorio Swift ci darà un controllo piccolo e spiegabile da riusare quando il sistema diventerà più ricco.

Entreremo quindi nei moduli e nella concorrenza, nel collegamento al modello e nel ciclo. Una risposta progressiva, un risultato tardivo e una conversazione che cambia ci aiuteranno a capire perché il programma debba conservare identità e stato. Seguiremo le classi e le funzioni reali dove dimostrano una responsabilità, lasciando al supplemento configurazioni ripetute e assemblaggio. Gli estratti manterranno collegamento con sorgente e revisione.

Le capacità verranno organizzate in un catalogo di strumenti con argomenti, risultati ed errori definiti. Una lettura e una modifica avranno conseguenze diverse; ripetere una richiesta non dovrà provocare involontariamente un secondo effetto. Le skill aggiungeranno istruzioni e risorse per compiti riconoscibili. Studieremo scelta, caricamento, applicazione, composizione e versioni delle procedure, fino ai conflitti fra indicazioni. Una skill guida il metodo del modello; il programma conserva il controllo delle operazioni ammesse.

La continuità richiederà di rappresentare obiettivi, durata, criteri e azioni consentite. Il modello potrà proporre un'organizzazione del lavoro; il programma conserverà gli impegni accettati e le condizioni di ripresa. Seguiremo deposito locale, eventi, agenda, attesa della persona, correzione e cancellazione. Il riferimento a Jarvis acquista qui un

significato concreto: mantenere un impegno non richiede che il modello continui a essere interrogato durante tutta l'attesa.

Distingueremo il ciclo del compito, nel quale una scelta produce un'osservazione, dal ciclo del servizio, che conserva lavoro e condizioni per riattivarlo. Un riavvio non trasforma un effetto incerto in un'operazione sicuramente mai avvenuta. La ricevuta di un servizio, lo stato locale e la situazione della risorsa possono non essere aggiornati nello stesso istante. Capiremo quali fatti permettano di riprendere, quali richiedano una nuova lettura e quando coinvolgere una persona.

Conoscenza e memoria ci faranno distinguere ciò che il modello riceve da ciò che l'applicazione conserva. Studieremo cronologia, contesto, memoria generale, taccuino e recupero di esperienze. Le preferenze manterranno origine e possibilità di correzione. I documenti saranno scelti per utilità e accessibilità; la somiglianza fra testi non basterà a dimostrarne la pertinenza. Le sintesi dovranno mantenere condizioni e incertezze anche quando riducono il materiale della richiesta.

Quando più partecipanti lavoreranno sullo stesso incarico, costruiremo il modello di una memoria operativa condivisa. La chiameremo anche lavagna: uno spazio governato nel quale evidenze, ipotesi, piani, autorizzazioni ed esiti mantengono ruoli diversi. Due aggiornamenti concorrenti non dovranno cancellarsi senza che il programma se ne accorga. Provenienza, revisioni e conflitti serviranno a consegnare a ciascun partecipante informazioni valide e ammesse per il proprio lavoro.

Su queste basi confronteremo sequenze, scelta del percorso, parallelismo, revisione e delega. Un coordinatore potrà dividere un incarico e un valutatore suggerire una correzione. Studieremo risposte mancanti e specialisti in disaccordo. L'agente singolo resterà un termine di confronto: aggiungere partecipanti deve affrontare un limite riconoscibile e comporta nuovi costi e nuove regole per ricomporre il lavoro.

Velia è anche un'applicazione usata da una persona. Conversazioni, finestre, personaggio, allegati, browser, Terminale e collegamenti con il Mac fanno parte dell'esperienza. Il personaggio dà un volto all'assistente; la sua espressione non dimostra una capacità tecnica o un'esperienza soggettiva. Per ogni canale seguiremo i componenti realmente disponibili nella versione descritta, dichiarando gli sviluppi ancora da costruire.

La sicurezza accompagnerà tutto il percorso. Un documento recuperato non dovrà modificare i permessi; una nota non renderà autorevole un'ipotesi; una skill esterna non concederà accessi. Per verificare questi confini useremo dati curati e ambienti controllati. La spiegazione del modello sarà un oggetto da valutare, mentre gli effetti verranno cercati nelle risorse che li possiedono.

Nella parte conclusiva ricomporrò un incarico lungo con interruzione, fonte contraddittoria, evento duplicato e correzione della persona. Seguiremo raccolta delle informazioni, confronto delle ipotesi, piano, eventuale autorizzazione, azione e verifica. Il trasferimento a un altro dominio, compreso il laboratorio Ordini e Coda, chiederà di scegliere dove basti una procedura e dove una scelta del modello aiuti davvero.

Dal nucleo di Velia alle nuove capacità

Una capacità mancante può richiedere un programma nuovo. Studieremo il passaggio da tool esistente e script di una skill a codice generato durante il lavoro: contratto, sorgente identificato, ambiente, compilazione o interpretazione, esecuzione, verifica, correzione e arresto. Una terminazione riuscita non dimostra che il calcolo sia corretto. Un processo separato o un ambiente di dipendenze non forniscono automaticamente l'isolamento necessario.

I confronti con LangChain4j e Spring AI conserveranno i temi del laboratorio Java e li metteranno in relazione con il nucleo Swift, senza presumere che API dai nomi simili abbiano le stesse proprietà. Incontreremo MCP, Model Context Protocol, per collegare capacità e informazioni; A2A, Agent2Agent, per scambiare incarichi e risultati con agenti indipendenti. I protocolli non sostituiranno le regole applicative su memoria, identità, accessi e validità del lavoro.

Quando più processi potranno interrompersi separatamente, approfondiremo riconciliazione degli effetti, messaggi, tentativi e compensazione. Il confronto con Temporal e Camunda renderà visibili altre forme di continuità e di attività umane. Ogni integrazione dichiarerà ambiente e confine effettivi; non attribuiremo a Swift una libreria nativa soltanto perché il problema è comune.

Affronteremo infine osservabilità, tempi, costi, distribuzione, migrazioni e miglioramento governato. Cambiare un modello, correggere una skill e aggiornare un ricordo sono modifiche diverse; ognuna richiede criteri e controlli pertinenti. Appendici, glossario, bibliografia annotata, esercizi e sorgenti accompagneranno il percorso.

Il libro crescerà per capitoli e raccolte progressive. Ogni uscita avrà un punto di arrivo comprensibile e materiali identificabili. Seguire gli avanzamenti di Velia ci permetterà di tornare sulle scelte con nuove domande, mantenendo distinguibili ciò che il codice prevede, ciò che una prova circoscritta mostra e ciò che rimane aperto.

Cominciamo dalla richiesta che vogliamo affidarle.

Fonti del racconto e delle prove

Le fonti seguenti permettono di ritrovare il metodo e le ricostruzioni richiamate nell'apertura. Le consultiamo insieme perché osservano parti differenti della vicenda.

Zhun Wang e altri, *ExploitGym*, 11 maggio 2026. Metodo del benchmark e condizioni delle prove.

OpenAI, *The Hugging Face incident and the road ahead*, 26 agosto 2026, e relativo *rapporto tecnico*. Contesto delle valutazioni, infrastruttura e ricostruzione dell'organizzazione coinvolta.

Hjalmar Wijk, Ajeya Cotra e Ryan Greenblatt, *indagine METR e Redwood Research*, 26 agosto 2026. Comportamenti e collaborazione, con limiti dell'analisi dichiarati.

Hugging Face, *Anatomy of a Frontier Lab Agent Intrusion*, luglio 2026. Ricostruzione degli accessi e dell'impatto dal lato del servizio attaccato.

1. Che cosa volevo da Velia

Quando penso a Velia, penso a un'assistente alla quale poter affidare un lavoro e con la quale poter capire a che punto siamo. Una conversazione è l'inizio di quel rapporto. Il problema diventa interessante quando la richiesta incontra documenti, applicazioni, accessi e attese: in quel momento le parole devono trovare un collegamento con ciò che il programma ha realmente fatto.

Possiamo osservare il passaggio attraverso una richiesta comune: confrontare due documenti e preparare una nota delle differenze. È abbastanza semplice da poter essere seguita dall'inizio alla fine, ma contiene già domande decisive. Il programma deve trovare le fonti corrette, leggerle, mantenere distinguibili le versioni, interpretare ciò che cambia e produrre un risultato che una persona possa controllare. Se serve anche un file, deve occuparsi della consegna.

In questo capitolo costruiremo il problema prima della soluzione. Il caso documentale è preparato per il libro; le responsabilità che ne ricaviamo ci guideranno nella lettura del progetto reale. Alla fine sapremo formulare un incarico, distinguere i suoi criteri di riuscita e riconoscere quando il risultato rimane incompleto. Ci servirà prima di scegliere classi, librerie e numero di agenti.

1.1 Una richiesta piccola, un lavoro che attraversa più confini

Immaginiamo una cartella chiamata accordi-fornitura, con due documenti: una proposta e la sua revisione. Chiediamo a Velia di confrontare tempi di consegna e assistenza, citare i passaggi pertinenti e preparare una nota. Per il momento il lavoro è in sola lettura: non deve cambiare i documenti o inviare la nota a qualcuno.

Questa richiesta contiene un oggetto, un risultato e dei vincoli. L'oggetto sono due documenti identificabili, non qualsiasi testo sullo stesso argomento. Il risultato è un confronto sostenuto dai passaggi letti. I vincoli limitano le azioni e stabiliscono come comportarsi se il materiale non basta. Dire «confronta tutto» senza chiarire ciò che deve entrare nella consegna lascia al programma troppe decisioni implicite.

Prepariamo un esempio. Nella prima proposta il passaggio sulla consegna dice «La consegna avviene entro dieci giorni dalla conferma dell'ordine». Nella revisione dice «La consegna avviene entro quindici giorni dalla conferma dell'ordine». Sono frasi inventate per il laboratorio. La differenza dei numeri è facile da riconoscere; il punto più importante è che entrambe misurano il termine a partire dallo stesso evento. Se il

secondo documento dicesse «dall'avvenuto pagamento», il confronto richiederebbe di spiegare anche il cambiamento della condizione.

Questo dettaglio mostra perché una sintesi breve possa essere precisa oppure fuorviante. «La consegna passa da dieci a quindici giorni» conserva il risultato soltanto se il riferimento temporale rimane uguale. Se cambia, il lettore deve saperlo. Non basta estrarre due valori: occorre capire a quale obbligo e a quali condizioni appartengano.

Esempio - Che cosa stiamo preparando

L'incarico potrebbe essere espresso così: «Confronta la proposta A e la revisione B per tempi di consegna e assistenza. Riporta le differenze con i passaggi che le sostengono. Se una parte non è leggibile, indica la lacuna. Prepara qui una nota, senza modificare i documenti e senza inviarla». È una richiesta didattica, non la trascrizione di un uso storico di Velia.

Il linguaggio rimane naturale. Non chiediamo all'utente di conoscere nomi di strumenti o comandi interni. L'assistente deve interpretare l'obiettivo; l'applicazione deve collegarlo a risorse e operazioni identificabili. Se la cartella non è ancora concessa o esistono più documenti candidati, quel dubbio richiede una scelta esplicita o una domanda. Il nome plausibile di un file non sostituisce l'identità della fonte.

1.2 Prima di dire «fatto», decidiamo che cosa deve essere vero

Nel libro *Dal prompt al software* abbiamo collegato il bisogno ai criteri di accettazione. Un criterio di accettazione descrive un risultato osservabile: permette di decidere se il comportamento ottenuto soddisfa la richiesta, senza imporre subito la forma del codice. Qui lo stesso metodo riguarda un lavoro che comprende più letture e più decisioni.

Per il confronto non basta che esista un testo finale. Dobbiamo riconoscere quali documenti sono entrati nel lavoro, quale versione è stata letta e se entrambe le aree richieste siano state affrontate. Le affermazioni devono avere riferimenti utilizzabili. Le parti mancanti devono restare riconoscibili. Il programma non deve aver modificato o inviato risorse escluse dal mandato.

Possiamo esprimere questi criteri in poche frasi, ma ognuna apre una responsabilità. «Indicare la versione» richiede un'identità conservata insieme alla lettura. «Segnalare una parte illeggibile» richiede che l'errore non venga perso nel passaggio al modello. «Non inviare» richiede che gli strumenti di invio siano esclusi o che la richiesta

attraversi un controllo capace di negarla. Un'istruzione nel testo non realizza da sola quel confine.

La tabella mette a confronto richiesta, fatto osservabile e limite del controllo. Serve a evitare che una buona prova di un dettaglio venga scambiata per la verifica dell'intero incarico.

Aspetto	Che cosa osserviamo	Che cosa non possiamo ancora concludere
Fonti	Le letture indicano documenti e revisioni usati.	Che ogni passaggio sia stato interpretato correttamente.
Copertura	Le sezioni richieste sono lette oppure dichiarate mancanti.	Che la sintesi ne abbia conservato tutte le condizioni.
Citazioni	Il riferimento conduce a un passaggio presente nella fonte indicata.	Che il passaggio sostenga la frase che lo cita.
Azioni	Le risorse modificate corrispondono alle operazioni ammesse.	Che il contenuto prodotto sia utile al destinatario.
Consegna	Il testo è disponibile nel luogo richiesto; se previsto, il file è stato salvato.	Che il solo annuncio del modello provi il salvataggio.

La terza riga è particolarmente importante. Una citazione esatta può essere collegata a una conclusione errata. Il documento potrebbe parlare di quindici giorni e il rapporto dichiarare dieci, pur riportando correttamente il passaggio fra virgolette. Il programma può verificare che le parole esistano; per valutare il rapporto fra quelle parole e la conclusione serve un controllo ulteriore. Il laboratorio del capitolo 4 renderà visibile questo limite.

1.3 La persona, il modello e l'applicazione

Per capire che cosa dobbiamo costruire separiamo tre responsabilità. La persona esprime il bisogno, rende disponibili le risorse e può chiarire condizioni che il sistema non conosce. Il modello interpreta richiesta e informazioni ricevute e propone come proseguire. L'applicazione conserva il lavoro e realizza le operazioni attraverso strumenti controllati.

Il modello non vede automaticamente una cartella perché il suo percorso compare nella conversazione. Un componente deve risolvere quel riferimento, controllare l'accesso e leggere i dati. Nel codice di Velia queste operazioni attraversano responsabilità diverse: il coordinatore organizza il turno, il nucleo conserva esecuzioni e passi, gli adattatori operano sulle risorse. Vedremo questi nomi quando ci aiuteranno a seguire un collegamento; per ora conta il fatto che nessuno di questi passaggi sia implicito nella frase dell'utente.

Il risultato di uno strumento torna poi al modello. Se la prima lettura segnala che l'assistenza è descritta in un allegato, l'assistente può cercare quel materiale, se disponibile e consentito. Se il programma riferisce che l'allegato manca, la nuova decisione deve usare questa informazione. L'assenza non deve scomparire perché il modello sa produrre una risposta ben formata.

La figura 1.1 segue questo circuito e distingue la decisione dall'effetto. Il punto di ritorno dal risultato al modello rende possibile scegliere il passo successivo usando ciò che è accaduto.

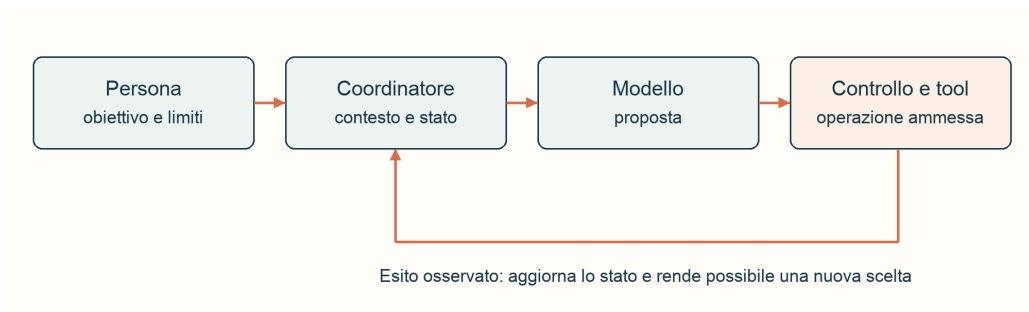


Figura 1.1 - La proposta attraversa controlli applicativi prima di diventare un'operazione. L'esito osservato alimenta il passo successivo; la persona riceve una consegna della quale devono essere riconoscibili fonti e limiti.

Questa ripartizione ci evita due errori. Il primo è affidare al modello anche l'autorità di decidere i propri permessi. Il secondo è programmare una lista di frasi e considerarla comprensione del bisogno. Velia mantiene una scelta diversa: il significato della richiesta viene interpretato dal modello, mentre identità, argomenti, revisioni e accessi sono controllati dal programma.

Definizione - Mandato e accesso

Il mandato descrive il lavoro ricevuto: obiettivo, limiti e risultato atteso. Un accesso consente un'operazione su una risorsa, per esempio leggere una cartella. Avere accesso a una cartella non impone di leggerla in ogni incarico; avere un incarico che la riguarda non crea automaticamente il permesso. Il programma deve mantenere entrambe le informazioni.

1.4 I cinque passaggi nascosti dietro «leggere i documenti»

Nel caso documentale dobbiamo poter distinguere almeno inventario, selezione, acquisizione, elaborazione e verifica. L'inventario ci dice quali elementi siano stati trovati. La selezione identifica ciò che serve al compito. L'acquisizione rende disponibile un contenuto leggibile. L'elaborazione mette in relazione i passaggi con la domanda. La verifica confronta la consegna con i criteri.

Questi passaggi possono sovrapporsi nell'implementazione, ma non hanno lo stesso significato. Un elenco di file non prova una lettura. Una lettura di byte non prova che l'estrazione abbia incluso correttamente una tabella. Un testo estratto non prova che il modello abbia ricevuto tutte le sue parti. Una risposta del modello non prova che abbia rispettato ogni condizione della richiesta.

In Velia l'inventario delle cartelle conserva riferimenti e informazioni sulle revisioni e distingue tipi di copertura. È un punto di lettura concreto: `FolderInventory.swift`, nel modulo `desktop`. Il suo contratto separa ciò che è trovato, acquisito e restituito. Qui usiamo questa distinzione per capire il problema; nei capitoli sulle cartelle seguiremo il codice e le condizioni della ripresa.

Supponiamo che il secondo documento contenga dodici pagine e che l'estrattore ne restituisca undici. Il confronto della consegna può essere corretto se il passaggio pertinente è presente; il confronto dell'assistenza potrebbe rimanere incompleto se la pagina mancante ne contiene le condizioni. Il programma deve riferire la lacuna con un ambito utile. Dire soltanto «analisi parziale» aiuta poco se la persona non sa quale conclusione ne sia toccata.

La formulazione della consegna deve quindi cambiare con ciò che è disponibile. Possiamo confermare la differenza dei tempi sulla base dei passaggi letti e dichiarare di non poter chiudere il confronto dell'assistenza. Il lavoro ha prodotto un risultato limitato ma utile. Diventa pericoloso quando la parola «completo» nasconde quel limite.

1.5 Interrompere cambia la domanda successiva

Ora immaginiamo di fermare il lavoro dopo la lettura della proposta A. Quando torniamo, scriviamo «Possiamo andare avanti con il confronto?». L'assistente deve capire a quale incarico ci riferiamo, ma il programma deve anche sapere che cosa possa recuperare e quali operazioni siano state realmente completate.

La cronologia della conversazione può contenere la richiesta. Il deposito operativo può contenere il tentativo interrotto e alcuni suoi passi. Un taccuino può riportare «manca B». Sono informazioni diverse. Il modello deve ricevere una vista pertinente di ciò che il programma conserva; il semplice fatto che esista un archivio non significa che ogni suo dato sia già nel contesto.

La distinzione si vede nei sorgenti di Velia: `ChatRuns.swift` riguarda le esecuzioni e la storia del contesto; `ChatNotebook.swift` descrive appunti con sezioni e basi diverse, compresa l'ipotesi. Non dobbiamo conoscere ancora tutti i tipi Swift. Basta osservare che l'applicazione attribuisce ruoli differenti alle informazioni e che la ripresa deve rispettarli.

Se B è cambiato durante l'interruzione, un vecchio appunto non prova quale sia il contenuto attuale. Se una nota potrebbe essere stata creata prima dell'arresto, ripetere l'operazione può duplicarla. Riprendere richiede quindi di ricostruire una situazione, distinguendo ciò che sappiamo da ciò che dobbiamo accertare. Il capitolo 3 userà proprio questo problema per leggere una modifica reale dell'architettura.

Approfondimento - Ricordare e rifare

Recuperare il testo di un tentativo precedente rende disponibile un'informazione. Rieseguire una chiamata produce una nuova operazione. Le due cose non sono equivalenti. Un estratto che dice «ho creato la nota» rimane una dichiarazione finché non viene confrontato con lo stato della risorsa o con una ricevuta valida. Il recupero non eredita automaticamente accessi o autorizzazioni.

1.6 Un'assistente completa richiede anche una buona esperienza d'uso

Il progetto non termina nel ciclo interno. La persona deve riconoscere quando Velia sta aspettando un risultato, quando chiede un chiarimento e quando il lavoro è stato fermato. Deve poter trovare le conversazioni e correggere una richiesta senza confondere la nuova versione con quella precedente. Queste sono responsabilità dell'applicazione e della sua interfaccia.

Il personaggio di Velia contribuisce all'identità e all'interazione. Le sue forme e le sue espressioni appartengono alla rappresentazione dell'assistente. Per capire una capacità dobbiamo però seguire i componenti che la realizzano. Un'animazione non prova che una fonte sia stata letta; un segnale di attività non prova che il servizio esterno stia avanzando.

La distribuzione aggiunge altri confini. Un componente presente sul computer dello sviluppatore potrebbe mancare su quello del lettore. Un accesso concesso a una build può richiedere di essere riesaminato in un'altra. Un servizio può supportare il formato richiesto soltanto in certe condizioni. Quando ricostruiremo l'applicazione, ambiente e dipendenze avranno lo stesso peso del codice che invia il prompt.

È qui che il caso specifico diventa interessante. La costruzione completa ci costringe a collegare qualità della risposta, integrità dello stato, esperienza della persona e comportamento delle risorse. Nessuno di questi aspetti può essere dedotto dal buon funzionamento di un altro.

1.7 Una variante che cambia davvero il problema

Modifichiamo la richiesta: oltre alla nota, la persona vuole che venga inviata al responsabile del progetto. La parte interpretativa può essere identica, ma cambia il tipo di effetto. Preparare un testo nella conversazione e inviarlo a un destinatario esterno richiedono accessi, identità e controlli diversi. Questo esempio descrive un'estensione del problema; non dichiara disponibile un canale di invio nella versione di Velia esaminata.

La variante chiede di decidere quali parti della soluzione riusare. Possiamo mantenere identificazione delle fonti, confronto e citazioni. Dobbiamo aggiungere un contratto per il destinatario, il contenuto esatto e la decisione che autorizza l'invio. Dopo l'azione dobbiamo sapere che cosa significhi la risposta del servizio: richiesta accettata e consegna avvenuta potrebbero essere fatti diversi.

Se il collegamento si interrompe, non basta ripetere. Prima occorre capire se esista un identificatore dell'operazione e se il servizio permetta di accertarne lo stato. In assenza di quel riscontro potremmo dover chiedere una decisione umana. La soluzione della lettura non si trasferisce integralmente alla scrittura: conserva il ragionamento sulle fonti, ma richiede un nuovo criterio sugli effetti.

Proviamo allora un esercizio. Riprendete l'incarico iniziale e sostituite «prepara qui una nota» con «salva un PDF nella cartella scelta». Descrivete che cosa debba essere vero per dichiarare la consegna e che cosa fare se la cartella non sia scrivibile. La risposta deve distinguere almeno contenuto pronto, documento convertito e file

salvato. Se cambia una fonte dopo la conversione, occorre inoltre stabilire a quale revisione appartenga il PDF. Una soluzione ragionata è disponibile negli apparati di questa raccolta.

1.8 Il criterio da portare al capitolo successivo

Abbiamo trasformato un desiderio in un lavoro osservabile. Sappiamo quali risorse riguardi, quali operazioni escluda e quali differenze il programma debba conservare. Abbiamo anche trovato il limite di una verifica puramente formale: la presenza di una citazione non garantisce il sostegno della conclusione.

Possiamo ora scegliere come organizzare quel lavoro. Potremmo usare una chiamata al modello, una sequenza programmata o un ciclo nel quale il modello decide come proseguire. Il capitolo successivo confronterà queste forme sullo stesso incarico. Non cercheremo l'etichetta più avanzata: cercheremo il punto nel quale una scelta del modello sia utile e il controllo che ne renda riconoscibile l'esito.

Riferimenti per proseguire

Il collegamento metodologico è il capitolo «Dal bisogno ai criteri di accettazione» di *Dal prompt al software*. Sviluppa il passaggio da un'intenzione ai comportamenti verificabili, con esempi propri.

I componenti di Velia richiamati qui appartengono alla revisione identificata nella guida ai sorgenti della raccolta. La guida permette di ritrovare file e responsabilità senza trasformare il percorso di lettura in un atlante di nomi.

2. Un modello, una conversazione, un sistema agentico

Abbiamo definito il lavoro: confrontare due documenti per consegna e assistenza, sostenere le differenze con le fonti e dichiarare ciò che manca. Adesso dobbiamo decidere come organizzarlo. Il fatto che Velia impieghi un modello linguistico non risolve questa scelta. Un modello può comparire dentro architetture molto diverse e svolgere in ciascuna un ruolo differente.

Partiremo dallo stesso incarico e cambieremo soltanto il modo in cui si scelgono i passi. Prima useremo una chiamata isolata; poi una procedura programmata; infine un ciclo che consente al modello di proporre il seguito dopo aver osservato un risultato. Il confronto ci permetterà di riconoscere che cosa renda agentic una parte del sistema e quando quella libertà sia giustificata.

2.1 Il modello lavora sul contesto che riceve

Un modello linguistico produce una risposta a partire da informazioni ricevute nella richiesta. Fra queste possono esserci istruzioni, messaggi, estratti di documenti e descrizioni degli strumenti disponibili. Chiamiamo contesto l'insieme di questi materiali per quella chiamata. Non coincide automaticamente con il contenuto del computer o con l'intera storia dell'applicazione.

Nel nostro caso potremmo inviare le due frasi sui tempi di consegna e chiedere un confronto. Il modello avrebbe già ciò che serve a spiegare il passaggio da dieci a quindici giorni, se le condizioni fossero uguali. Qualcun altro avrebbe però individuato e letto le fonti. La capacità di acquisire il materiale non è contenuta nella sola operazione che genera il testo.

La generazione è probabilistica: il modello usa i parametri appresi per attribuire preferenze alle continuazioni possibili. Questo non significa che ogni frase sia scelta casualmente né che due richieste debbano sempre produrre risposte diverse. Significa che la correttezza del risultato non deriva da una regola applicativa che garantisca il significato di ogni frase. Parametri di generazione, servizio e contesto influenzano il comportamento, ma il controllo della consegna rimane necessario.

Se il modello restituisce una struttura con il nome di uno strumento, abbiamo una proposta interpretabile dal programma. Un oggetto che contiene `leggi_documento` non

legge il documento da solo. L'applicazione deve riconoscere la capacità richiesta, controllarne gli argomenti e realizzare l'operazione. Soltanto il risultato di quel passaggio può diventare una nuova osservazione.

Concetto chiave - Un formato valido non rende vera una proposta

Una richiesta può rispettare perfettamente lo schema dei dati e riferirsi comunque alla fonte sbagliata. Il controllo del formato risponde a «posso interpretare questa struttura?». Il controllo del compito risponde a «questa operazione è pertinente e ammessa adesso?». La valutazione del risultato chiede infine se ciò che abbiamo ottenuto sostenga davvero la conclusione.

2.2 Prima soluzione: una chiamata e una risposta

Nel primo approccio prepariamo due estratti e chiediamo al modello di confrontarli. Il programma potrebbe limitarsi a comporre la richiesta, inviarla e mostrare il testo ricevuto. Se le fonti sono brevi, complete e già note, questa soluzione è comprensibile e può essere adeguata.

La sua semplicità rende visibile anche il confine. Se manca il secondo estratto, il modello non lo acquisisce automaticamente. Può dichiarare che il confronto non è possibile, oppure può produrre una risposta non sostenuta. L'applicazione deve distinguere questi esiti. Se vogliamo che il programma chieda il documento mancante, stiamo già aggiungendo un'interazione e una condizione da governare.

Una conversazione conserva più turni, ma non rende da sola operativa l'acquisizione. La persona può leggere i documenti e incollare gli estratti, correggere un dettaglio o porre una domanda successiva. Il modello beneficia del contesto che gli viene rimandato. Il lavoro sulle risorse rimane però attribuito alla persona o a un altro componente, finché l'applicazione non realizza un collegamento con esse.

Per questo la presenza di una chat non basta a classificare il sistema. Una chat può essere l'interfaccia di una chiamata isolata, di una procedura o di un agente. Dobbiamo seguire il percorso interno: chi può chiedere un'operazione, chi la esegue e che cosa accade con il suo esito.

Nel caso didattico, il criterio per scegliere questa soluzione è concreto. Se ogni incarico arriva con esattamente due estratti verificati e la sola richiesta è formulare una spiegazione, la chiamata isolata affronta il problema senza bisogno di un catalogo di strumenti. Dobbiamo comunque controllare fedeltà e utilità del confronto. Se invece il materiale va cercato e cambia durante il lavoro, il confine non è più sufficiente.

2.3 Seconda soluzione: programmare il percorso

Nel secondo approccio il codice stabilisce le attività. Può leggere la proposta A, leggere la revisione B, estrarre le sezioni richieste e chiedere al modello di formulare il confronto. In caso di lettura fallita può produrre una consegna parziale. Questo è un workflow: un processo nel quale attività, condizioni e diramazioni sono definite dal programma.

Un workflow può essere articolato. Può ripetere una lettura entro limiti, aspettare una risposta della persona e scegliere un ramo sulla base di un dato. Può usare un modello in più punti. La sua caratteristica non è l'assenza di decisioni, ma il fatto che il percorso ammesso e i criteri delle diramazioni siano organizzati nel codice.

Per esempio, sappiamo che ogni documento contiene due sezioni identificabili e che entrambe vanno sempre acquisite. Non serve chiedere al modello quale leggere per prima, se quell'ordine non cambia il risultato e non risolve un'incertezza. Possiamo programmare le letture e riservare al modello il compito nel quale l'interpretazione linguistica è utile.

Il limite emerge quando il materiale rimanda a fonti ulteriori non tutte previste, oppure quando la domanda cambia in modo sostanziale da un incarico all'altro. Possiamo estendere il workflow, ma ogni nuova combinazione richiede di capire quali diramazioni aggiungere e come mantenerle. A un certo punto può essere utile lasciare al modello la proposta del passo successivo, mantenendo controlli su capacità e risorse.

Questa è una decisione di progettazione, non un passaggio obbligatorio verso un sistema migliore. Un percorso definito può offrire chiarezza e prove più dirette su un incarico stabile. Il confronto deve valutare se la flessibilità aggiunta affronti un limite reale, insieme ai suoi costi e ai nuovi errori possibili.

2.4 Terza soluzione: osservare e scegliere di nuovo

Nel ciclo agentico il modello riceve obiettivo, informazioni disponibili e operazioni che può proporre. Sceglie un passo; il programma lo controlla ed esegue; il risultato viene incluso nella richiesta successiva. Il modello decide allora come continuare usando ciò che ha appreso.

Torniamo al documento B. La prima lettura della sezione assistenza potrebbe indicare «si applicano le condizioni dell'allegato tecnico». L'assistente può proporre di cercare l'allegato, se l'operazione è disponibile e il mandato lo consente. Se l'allegato manca, il

risultato della ricerca deve tornare nel ciclo. La conclusione potrà spiegare la lacuna e chiedere il materiale necessario, anziché riempirla con una condizione plausibile.

La tabella segue un percorso preparato, con lo stesso obiettivo del capitolo precedente. Non è la trascrizione di una risposta reale di Velia e non misura la frequenza con cui un modello sceglierebbe questi passi.

Informazioni disponibili	Proposta possibile	Nuovo fatto da riportare al modello
Incarico e risorse concesse	Inventariare la cartella.	Sono presenti A e B; l’inventario indica identità e limiti.
Documenti identificati	Leggere le sezioni di consegna.	Sono disponibili i due passaggi con le rispettive revisioni.
Differenza dei tempi e assistenza da esaminare	Leggere l’assistenza di A e B.	I passaggi sono disponibili; B rimanda anche a un allegato tecnico.
Rinvio a un’altra fonte	Cercare l’allegato nel perimetro ammesso.	L’allegato non è disponibile.
Fonti acquisite e lacuna esplicita	Consegnare il confronto parziale e chiedere l’allegato.	I criteri di completezza rimangono parzialmente non soddisfatti.

L’aspetto decisivo è il ritorno dell’osservazione. Se il programma eseguisse una lista di azioni scritta una volta all’inizio, senza permettere di riesaminarla dopo un errore, avremmo una diversa organizzazione. Un piano può essere utile, ma deve restare collegato ai fatti che lo rendono valido. Chiedere al modello di dichiarare «continuerò con l’allegato» non garantisce che l’allegato sia accessibile.

La figura 2.1 affianca le tre forme. Le frecce di ritorno mostrano dove un risultato può influenzare una nuova scelta; nel workflow quelle scelte seguono le condizioni programmate, mentre nel ciclo il modello propone il seguito entro i confini applicativi.

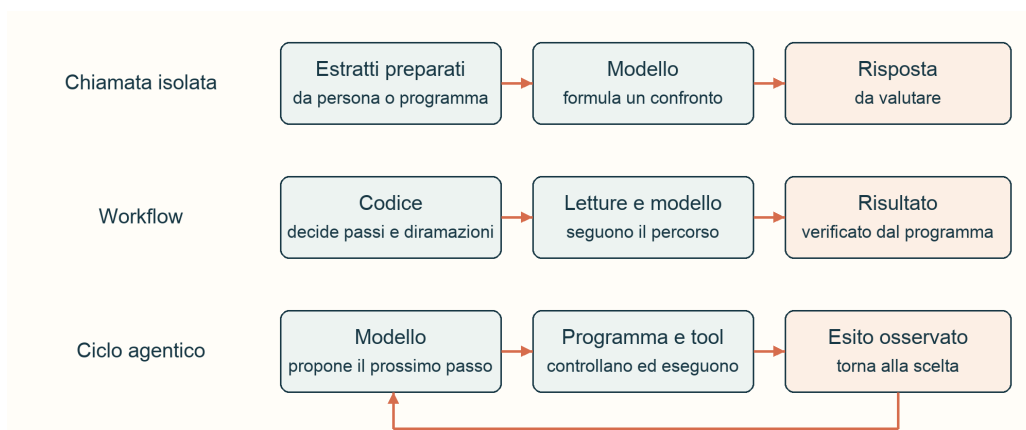


Figura 2.1 - La differenza riguarda come si decide il seguito. Una procedura può contenere modelli e diramazioni. Un agente aggiunge una scelta del modello sul percorso, senza eliminare i controlli del programma.

Riferimento essenziale - Una distinzione architetturale

In *Building effective agents*, Anthropic distingue percorsi prestabiliti e sistemi in cui il modello guida dinamicamente il processo. È una distinzione utile per leggere le responsabilità; non è una prova che l'agente sia sempre preferibile. Qui la applichiamo allo stesso incarico per confrontare decisioni, esiti e costo della complessità.

2.5 Chi decide, chi controlla, chi può fermare

Nel percorso principale di Velia l'interpretazione della richiesta appartiene al modello. Il codice non deve trasformare una lista di parole come «confronta», «continua» o «salva» in un sostituto di quella comprensione. Una frase può contenere il nome di un'operazione proprio per escluderla: «non salvare ancora, spiegami soltanto le differenze».

Restano invece necessari controlli deterministici, cioè verifiche con una regola esplicita sul dato ricevuto. Il programma può controllare se un identificatore corrisponde a una risorsa, se una versione è ancora valida, se il nome di un tool appartiene al catalogo e se l'operazione rientra negli accessi assegnati. Questi controlli non cercano di ricostruire il significato della frase naturale: operano sulla proposta e sullo stato del sistema.

Il programma può quindi rifiutare un'azione anche se il modello la considera utile. Il rifiuto deve diventare un esito comprensibile per il ciclo. Se il modello continua a chiedere una capacità indisponibile, il sistema ha bisogno di condizioni di arresto e di

una consegna che spieghi il limite. La presenza di una nuova decisione non giustifica una ripetizione senza termine.

L'arresto può avere cause diverse. Il lavoro può essere soddisfatto, mancare di dati, avere esaurito una risorsa o essere stato annullato. Questi stati non dovrebbero essere riuniti sotto una generica etichetta «finito». Per la persona e per la ripresa del programma significano cose differenti.

2.6 Una sola parola non descrive tutta l'autonomia

Quando diciamo «agente autonomo» possiamo riferirci a decisioni diverse. L'assistente può scegliere quali fonti leggere, ma dover chiedere conferma per una modifica. Può riprendere un incarico a una scadenza senza poter creare nuovi obiettivi. Può usare molti strumenti restando limitato a una sola cartella. Il numero di capacità non misura da solo l'autonomia.

Separiamo quindi il percorso decisionale, la durata, gli accessi e il coordinamento. Il percorso descrive quali passi il modello può proporre. La durata riguarda la continuità dell'incarico oltre una singola risposta. Gli accessi delimitano risorse ed effetti. Il coordinamento stabilisce se il lavoro appartenga a un agente oppure coinvolga partecipanti distinti.

Una pianificazione periodica non implica che il modello sia sempre attivo. Il programma può conservare una scadenza e preparare una richiesta soltanto quando serve. Una memoria persistente non implica iniziativa: si possono conservare dati e non avviare alcun lavoro fino a una nuova richiesta. Un sistema con più agenti non è automaticamente più autonomo di uno con un solo agente.

Queste distinzioni orientano anche la valutazione. Se vogliamo capire se la delega aiuti, dobbiamo confrontarla con un singolo agente in condizioni comprensibili. Se cambiamo contemporaneamente modello, strumenti, memoria e tempo disponibile, diventa difficile attribuire la differenza a un fattore. Possiamo osservare un risultato utile, ma la sua spiegazione causale richiede un confronto progettato per quella domanda.

2.7 La memoria del sistema e i parametri del modello

Ogni giro del ciclo riceve una vista delle informazioni disponibili. Il fatto che l'applicazione abbia salvato un documento non garantisce che il modello lo riceva al giro successivo. Occorre recuperare il contenuto pertinente e includerlo nella richiesta, con il suo ruolo e i suoi limiti.

La cronologia conserva messaggi. Lo stato del compito rappresenta obiettivi, operazioni ed esiti. Un taccuino può conservare appunti e questioni aperte. La memoria generale può conservare preferenze riusabili fra conversazioni. Una base documentale permette di recuperare conoscenza da fonti. Possono usare archivi diversi o condividere un'infrastruttura; restano funzioni differenti.

Il modello di base mantiene i parametri appresi nell'addestramento. Nel normale recupero di un appunto cambia il contesto disponibile all'inferenza, non si presume una nuova fase di addestramento. Questo rende utile parlare con precisione di «imparare»: Velia può conservare una correzione o una procedura migliore senza che i parametri del modello siano cambiati.

La distinzione previene anche un errore di autorità. Un ricordo può dire che la persona preferisce note concise. Non può concedere il diritto di inviarle a chiunque. Per decidere un'operazione servono il mandato corrente e i controlli degli accessi. I dati della memoria possono essere informativi, inesatti o superati; il programma deve mantenere il confine.

2.8 Quanto costa poter scegliere di nuovo

Una chiamata ulteriore richiede tempo e risorse. Per capire il costo del ciclo usiamo un conto semplice, con numeri inventati a fini didattici. Supponiamo un percorso strettamente sequenziale: tre chiamate al modello, ciascuna di 0,8 secondi, e due operazioni locali, ciascuna di 0,1 secondi. Ignoriamo per ora tempi dell'interfaccia, preparazione e rete non già compresi.

Indichiamo con T il tempo totale, con M il tempo di una chiamata al modello e con U quello di un'operazione. Tutti i tempi sono espressi in secondi. In questo caso, nel quale attività e durate sono quelle dichiarate, il conto è:

$$T = 3 \times M + 2 \times U$$

$$T = 3 \times 0,8 \text{ s} + 2 \times 0,1 \text{ s} = 2,6 \text{ s}$$

Il risultato torna: le tre chiamate richiedono 2,4 secondi e le due operazioni 0,2 secondi. Non è una misura delle prestazioni di Velia. Se una lettura può procedere in parallelo con un'altra, se i tempi variano o se il servizio richiede più tentativi, la formula va adattata. Serve soltanto a rendere visibile che una nuova scelta ha un costo e che il suo vantaggio deve riguardare il lavoro.

Anche il costo monetario dipende da modello, servizi e materiale elaborato; le condizioni vanno controllate al momento dell'uso. Il criterio trasferibile è separare numero di decisioni, dati forniti, durata e qualità del risultato. Ridurre le chiamate

può peggiorare un compito che richiede una nuova osservazione; aggiungerle può peggiorare un compito già risolto da una procedura breve.

Approfondimento - Un'osservazione può cambiare il percorso

Il lavoro *ReAct*, di Shunyu Yao e colleghi, studia l'alternanza fra elaborazione e azioni che acquisiscono informazioni dall'ambiente. Ci aiuta a riconoscere il ruolo dell'osservazione. I risultati dello studio riguardano i compiti esaminati; non misurano l'affidabilità di Velia né rendono una spiegazione generata dal modello una prova del suo processo interno.

2.9 Scegliere la forma adatta al problema

Proviamo una variante. Ogni notte dobbiamo confrontare gli stessi due file e verificare cinque campi con formato stabile. Se un campo manca, il programma deve segnalarlo; il modello serve soltanto a scrivere una nota leggibile. Una procedura programmata ha un motivo concreto: il percorso è noto e i controlli hanno condizioni esplicite.

Cambiamo ora il bisogno: la persona vuole ricostruire quali documenti spieghino una differenza, seguire riferimenti incrociati e porre domande quando il materiale non basta. La scelta del seguito diventa parte sostanziale del lavoro. Un ciclo agentico può essere una soluzione, purché strumenti, accessi e criteri rendano controllabili i passi. La promessa deve essere verificata su casi che rappresentino quella variabilità.

L'esercizio consiste nel formulare un caso per ciascuna architettura e spiegare il motivo della scelta. Non basta scrivere «più flessibile». Occorre indicare una decisione che cambia in funzione di un'osservazione e un controllo che permetta di riconoscere l'errore. Per il caso agentico aggiungete anche una condizione di arresto. Le soluzioni ragionate della raccolta propongono esempi e limiti.

Abbiamo ora una distinzione che useremo nella lettura del codice. Il capitolo successivo mostrerà che cosa cambia quando una lista di parole tenta di decidere se la persona voglia riprendere un vecchio incarico, e come quella responsabilità possa essere riportata al modello mantenendo i controlli del programma.

3. Sviluppare Velia con l'aiuto dell'AI

Costruire Velia significa usare l'intelligenza artificiale in due lavori diversi. Nel primo, un assistente di programmazione aiuta a leggere il progetto, proporre modifiche e scrivere codice. Nel secondo, il modello collegato a Velia interpreta l'incarico della persona e propone operazioni. Se non distinguiamo questi piani, rischiamo di accettare una buona spiegazione del codice come prova di un buon comportamento dell'applicazione.

Per seguire il metodo useremo un cambiamento reale. Nella storia del progetto il meccanismo che recupera un tentativo interrotto è stato modificato: la selezione basata su parole iniziali della richiesta ha lasciato posto a un candidato individuato dal programma e a una decisione di pertinenza affidata al modello. Non ricostruiremo una conversazione di sviluppo che non possediamo. Leggeremo le versioni, esplicheremo il problema e prepareremo i criteri con cui giudicare la soluzione.

Il cambiamento è identificato dal commit `a447093` del repository Jarvis, del 26 ottobre 2026. Il titolo del commit descrive l'intenzione di mantenere l'interpretazione guidata dal modello; la modifica comprende anche altri interventi. Qui esaminiamo soltanto il confine della ripresa in `ChatAgentCoordinator.swift`. La raccolta si riferisce poi alla revisione complessiva indicata nella guida ai sorgenti: la scelta di un episodio non equivale a una verifica di tutto il commit o del prodotto.

3.1 Il problema dietro «possiamo andare avanti?»

Il lavoro sui documenti è stato interrotto. La persona torna e scrive «Possiamo andare avanti con il confronto?». La nuova richiesta può riferirsi al tentativo precedente. Potrebbe però chiedere una spiegazione su un argomento diverso, pur usando la parola «continua». Dobbiamo decidere se leggere materiali del vecchio lavoro prima di introdurli nel nuovo contesto.

Il deposito può sapere che esiste un tentativo precedente e che è fallito o interrotto. Può sapere a quale conversazione appartenga e quali passi risultino completati. Non può ricavare il significato della nuova frase soltanto dalla presenza di quei dati. L'esistenza di un candidato e la pertinenza del candidato sono domande diverse.

Nella versione precedente il codice normalizzava la richiesta, conservava una lista di espressioni come «ripren-di», «continua», «resume» e richiedeva che l'obiettivo fosse

identico a quello passato oppure che la frase iniziasse con una delle espressioni. Lo schema si può leggere nell'estratto seguente, riformattato dalla sorgente storica:

```
guard objective == prior.objective
    || resumes.contains(where: { request.hasPrefix($0) })
else { return nil }
```

`objective` è il testo della richiesta corrente; `prior.objective` è l'obiettivo conservato del lavoro precedente. `resumes` contiene le formule preparate dal programma. `hasPrefix` verifica l'inizio della stringa. In Swift, `guard` richiede che la condizione sia vera per continuare; altrimenti esegue il ramo che qui restituisce l'assenza di un risultato.

Il controllo è facile da descrivere e ripetere, ma risponde a una domanda lessicale: la frase è uguale oppure comincia con una formula prevista? Il bisogno dell'utente riguarda invece il significato: questa richiesta vuole continuare proprio quel lavoro? Una lista più lunga può riconoscere altre formulazioni e conservarne comunque il limite.

Possiamo dimostrarlo con esempi ragionati, senza presentarli come esiti osservati sull'app. «Torniamo al confronto che stavamo preparando» può riferirsi chiaramente al tentativo pur non iniziare con una parola della lista. «Continua a spiegarmi la differenza fra i modelli, lasciamo perdere i documenti» comincia con una parola ammessa ma può indicare un altro obiettivo. Il problema è il rapporto fra richiesta e contesto, non soltanto il vocabolario.

Concetto chiave - Un controllo preciso può rispondere alla domanda sbagliata

Il riconoscitore può funzionare esattamente come scritto. Questo non dimostra che affronti correttamente il bisogno. Un test che confermi la presenza di «continua» verifica la regola lessicale; non dimostra la comprensione della volontà di riprendere un incarico. Prima dei test dobbiamo rendere esplicita la domanda alla quale il componente deve rispondere.

3.2 Leggere il contesto prima di chiedere una modifica

In *Dal prompt al software* abbiamo chiamato questo passaggio comprensione del progetto. Prima di formulare una richiesta all'assistente di programmazione occorre sapere chi usa la funzione, quali dati riceve, che cosa produce e quali controlli non devono andare persi. Il singolo frammento non basta.

Nel caso di Velia leggiamo il coordinatore che prepara la richiesta al modello, le funzioni del deposito che restituiscono la storia e il contratto dello strumento di

recupero. Leggiamo anche le istruzioni di progetto: l'interpretazione del linguaggio naturale appartiene al modello, mentre il programma verifica provenienza, identità, revisioni, parametri e accessi. Questa preferenza orienta la soluzione e impedisce di risolvere il problema aggiungendo un'altra lista di frasi.

Un assistente di programmazione al quale mostrassimo soltanto la condizione potrebbe proporre sinonimi, confronto senza distinzione fra maiuscole e minuscole o una ricerca in tutta la frase. Sarebbero modifiche coerenti con l'ipotesi che il componente debba classificare parole. Il contesto cambia la domanda: dobbiamo spostare la responsabilità semantica, mantenendo quella strutturale.

La storia del repository aiuta a rendere verificabile il confronto. Una differenza fra due revisioni, spesso chiamata diff, mostra quali righe siano state aggiunte o rimosse. È un materiale utile per capire la trasformazione, ma non prova da solo l'effetto in esecuzione. Un commento può dichiarare un'intenzione che il resto del percorso non rispetta. Per questo seguiamo anche il punto nel quale la funzione viene chiamata e il modo in cui il risultato entra nel contesto.

3.3 Formulare il bisogno senza scrivere in anticipo la risposta

Possiamo preparare una richiesta didattica all'assistente di programmazione. Non è il prompt storico usato dall'autore; mostra come trasferire obiettivo e vincoli senza scegliere già ogni dettaglio dell'implementazione.

Esempio - Una richiesta di sviluppo controllabile

«Esamina il recupero dei tentativi interrotti. La pertinenza della nuova richiesta deve essere valutata dal modello, non da parole chiave o dall'uguaglianza dei testi. Il codice deve individuare soltanto un candidato strutturalmente ammesso e verificare l'identità quando viene chiesto il recupero. Conserva i confini di conversazione, collaborazione, delega, accessi e consegna. Proponi la modifica più circoscritta e indica quali prove verifichino il programma e quali richiedano un modello reale».

La richiesta definisce il bisogno, il confine e il risultato atteso. Non impone il nome di ogni funzione né il numero di nuove classi. Consente di valutare alternative: possiamo confrontare responsabilità e complessità prima di accettare il codice proposto.

La revisione deve cercare anche ciò che la richiesta non rende immediatamente evidente. Leggere vecchi estratti può introdurre informazioni non pertinenti o superate. Un vecchio risultato non deve diventare un'istruzione autorevole. Il fatto

che una funzione restituisca testo non deve permettere di ripetere una vecchia azione o ereditare un permesso. Questi vincoli collegano memoria e sicurezza.

Se l'assistente propone di caricare automaticamente tutto il vecchio lavoro e lasciare al modello il compito di ignorarlo quando non serve, dobbiamo confrontare quella proposta con il bisogno. Abbiamo chiesto di valutare la pertinenza prima di leggere gli estratti, non soltanto prima di usarli nella risposta. La collocazione del controllo cambia quali dati entrano nel nuovo contesto.

3.4 La nuova separazione: trovare un candidato, decidere se serve

Nella revisione successiva troviamo una funzione chiamata `recoveryCandidate`. Il nome indica un candidato, non una decisione definitiva sulla richiesta. L'estratto seguente proviene dalla revisione del percorso adottata per il libro ed è riformattato per la lettura:

```
private func recoveryCandidate(
    sessionID: UUID, currentRunID: UUID
) async throws -> AgentRun? {
    guard let prior = try await store
        .chatContextHistory(sessionID: sessionID)
        .first(where: { $0.id != currentRunID }),
        [.failed, .interrupted].contains(prior.state)
    else { return nil }

    let previous = try await store.chatManifest(runID: prior.id)
    guard previous.collaborationID == nil,
        previous.delegationTicket == nil
    else { return nil }
    return prior
}
```

Il codice chiede al deposito la storia del contesto e considera il primo elemento diverso dall'esecuzione corrente. Verifica che lo stato sia fallito o interrotto. Legge poi i dati dell'esecuzione precedente e richiede che non appartenga ai percorsi di collaborazione o delega esclusi da questo recupero. Se una condizione manca, non offre un candidato. Non scorre arbitrariamente tutte le esecuzioni finché ne trova una che sembra utile.

UUID indica un identificatore; `AgentRun?` esprime che il risultato può essere un'esecuzione oppure essere assente. `async throws` segnala che la funzione può attendere operazioni e produrre un errore. Per ora ci serve comprendere questi ruoli, non tutti i dettagli della concorrenza Swift: li svilupperemo nel ponte e nei capitoli dedicati.

Il dato che conta è ciò che non entra nella condizione: non compare un elenco di parole dell'utente. Il programma verifica relazioni strutturali fra record. È una responsabilità che può essere descritta e testata senza chiedere al codice di interpretare una frase naturale.

Il coordinatore presenta quindi al modello informazioni sul candidato, compresi riferimento alla richiesta originaria e stato disponibile. Le istruzioni chiedono di valutare se la richiesta corrente riprenda quel lavoro, anche con parafrasi e lingue diverse. Se il candidato è pertinente, il modello può chiedere `chat_recovery`; se la richiesta è nuova, deve ignorarlo. Una richiesta originaria non recuperabile rimane una lacuna esplicita e può richiedere chiarimento.

Questa è una scelta semantica: riguarda il rapporto fra intenzione attuale e lavoro precedente. Le istruzioni la assegnano al modello; non garantiscono che il modello la compia sempre correttamente. Il programma conserva comunque il controllo dell'identità del candidato che può essere letto.

3.5 Lo strumento non accetta un passato qualsiasi

Quando il modello chiede il recupero, non consegna un accesso generico all'archivio. Il percorso valida gli argomenti strutturati e richiede l'identificatore del candidato selezionato. Prima di restituire gli estratti riesamina le condizioni. La situazione potrebbe essere cambiata fra la preparazione del contesto e la chiamata dello strumento.

Nella funzione `recoveryReference` troviamo il collegamento fra identità dichiarata e candidato ancora valido. Il risultato raccoglie soltanto il materiale ammesso dal percorso. Le regole distinguono tipi di passo e modalità precedenti; il codice conserva anche un confine sui testi grezzi di un tentativo Cyber fallito. Non estenderemo questa osservazione a una garanzia generale sulla modalità Cyber: seguiremo il controllo nel suo perimetro.

Il testo restituito dichiara il proprio ruolo: riferimenti al lavoro precedente, non istruzioni, permessi, prova di completamento o conferma dell'accesso attuale. Questa etichetta aiuta il modello, mentre la protezione delle operazioni continua a dipendere dai controlli applicativi. Se un estratto contiene una vecchia frase «documento salvato», il nuovo tentativo deve ancora accertare ciò che serve prima di ripetere un effetto.

La figura 3.1 mette in evidenza i due momenti. Il programma delimita i candidati e i dati recuperabili; il modello valuta la pertinenza; il programma controlla nuovamente

l'identità prima della lettura. Nessuna freccia rilegge ed esegue automaticamente la vecchia lista di azioni.

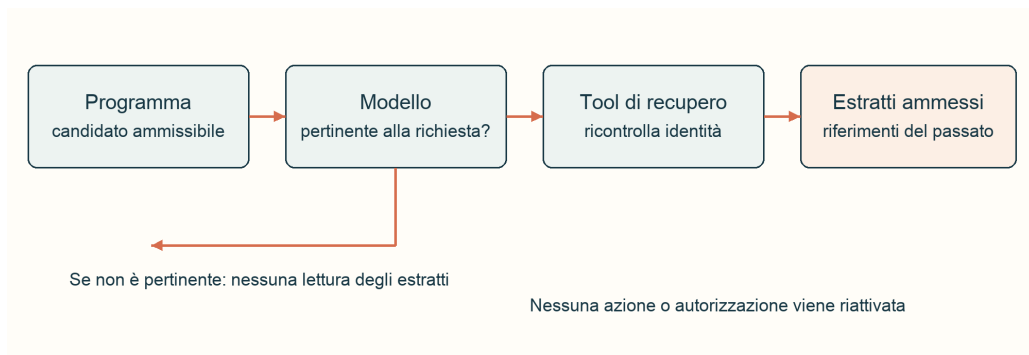


Figura 3.1 - La decisione sul significato è separata dall'ammissibilità della lettura. Il recupero restituisce informazioni del passato con limiti espliciti e non riattiva le autorizzazioni precedenti.

3.6 Le prove devono seguire le responsabilità

Per il programma possiamo preparare casi con una storia nota: nessuna esecuzione precedente, esecuzione precedente completata, tentativo interrotto ammesso, candidato appartenente a un percorso escluso. Possiamo chiedere il recupero con un identificatore diverso da quello proposto o cambiare le condizioni prima della lettura. In ciascun caso il risultato atteso deriva dal contratto strutturale.

Queste prove non hanno bisogno che un modello capisca «andiamo avanti». Possiamo controllare il programma con una scelta preparata. La loro utilità sta proprio nell'isolare il confine: se una chiamata con identificatore errato restituisce comunque gli estratti, il problema riguarda il controllo del dato, indipendentemente dalla qualità del modello.

La valutazione semantica richiede altri casi. Una richiesta può riprendere lo stesso lavoro senza parole della vecchia lista; può contenere «continua» e chiedere qualcosa di diverso; può essere ambigua. Il risultato atteso deve consentire anche una domanda, quando le informazioni non bastano. Un elenco di parole cercate nella risposta riprodurrebbe il problema che stiamo cercando di superare.

Per ogni valutazione con un modello reale occorre mantenere riconoscibili modello, configurazione, contesto, strumenti e criteri. Ripetere i casi aiuta a osservare la variabilità, ma non trasforma un campione in una garanzia assoluta. Qui abbiamo letto una trasformazione documentata del codice; non stiamo dichiarando una nuova campagna di valutazione del modello o la qualifica dell'applicazione.

Buona pratica - Due domande, due famiglie di prove

«Un identificatore non ammesso viene rifiutato?» può essere verificato con ingressi controllati. «Il modello distingue una ripresa da una nuova domanda?» richiede casi di significato e un modello effettivo. La prima prova non autorizza a rispondere alla seconda. Anche le istruzioni più precise rimangono condizioni del sistema, non evidenze del comportamento che vogliamo osservare.

3.7 Il codice proposto deve diventare comprensibile

L'aiuto dell'AI nello sviluppo ha valore quando permette di esplorare e correggere una soluzione mantenendo la responsabilità del risultato. Accettare un diff perché sembra coerente con la richiesta non basta. Dobbiamo poter spiegare perché il candidato venga individuato in quel punto, quali dati entrino nel contesto e quale controllo avvenga al momento dell'effetto.

Nel caso esaminato possiamo porre domande precise. La nuova funzione ha soltanto cambiato nome al riconoscitore? Le istruzioni al modello sono presenti nel percorso realmente usato? Il tool di recupero è disponibile soltanto quando il programma ha un candidato ammesso? L'identità viene verificata di nuovo? Le informazioni recuperate possono riaprire accessi? Ogni risposta deve trovare un riscontro nel codice pertinente o rimanere una domanda aperta.

La stessa revisione riguarda manutenzione e dipendenze. Se il cambiamento può essere realizzato con le strutture esistenti, una nuova libreria di classificazione richiede una motivazione ulteriore. Se crea un secondo modo di leggere la storia, dobbiamo confrontarlo con quello già usato. Il metodo di riuso non significa evitare qualsiasi dipendenza; significa sapere quale problema aggiuntivo risolva e quale costo introduca.

Le istruzioni di progetto e i criteri di accettazione aiutano anche l'assistente di programmazione. Rendono disponibili le decisioni che non dovrebbero essere inventate durante la generazione. Il programmatore deve comunque riesaminare la proposta, comprendere il codice e applicare prove adeguate. È il senso di «il codice deve diventare nostro», sviluppato nel capitolo conclusivo di *Dal prompt al software*.

3.8 Conservare ciò che permette di ripetere il controllo

Una verifica ripetibile ha bisogno di sorgenti, ingressi curati, configurazione e criteri. Non richiede di accumulare tutti gli intermedi di ogni esecuzione. Nel progetto i risultati dei test, i rendering di controllo, i report e le tracce delle prove vengono valutati durante il ciclo e poi rimossi. Il laboratorio del prossimo capitolo segue questa regola anche per il compilatore e le sue cache.

Gli ingressi curati, chiamati *fixture*, sono scelti per rappresentare un caso: un documento breve, una proposta preparata o uno stato del compito. Non sono la risposta dell'ultima esecuzione rinominata per conservarla. Una figura originale del libro è un asset editoriale; l'immagine prodotta per controllare l'impaginazione è un intermedio di verifica. Distinguere questi ruoli permette di mantenere riproducibilità e ordine.

I dati operativi di Velia hanno un altro scopo. Conversazioni, esecuzioni e documenti dell'utente possono servire al lavoro e seguono regole proprie di conservazione e cancellazione. La pulizia dei test non autorizza a cancellarli. Nel nostro laboratorio useremo soltanto cartelle e file preparati per la prova, senza leggere l'archivio operativo o le credenziali dell'applicazione.

3.9 Una variante da progettare

Immaginate che esistano due vecchi lavori entrambi interrotti e che la persona dica «riprendiamo quello sui contratti». La funzione letta in questo capitolo non è un motore generale che sceglie fra tutti i lavori: offre un candidato attraverso un percorso strutturale definito. La nuova richiesta può richiedere un'altra capacità, oppure un chiarimento.

L'esercizio chiede di progettare l'estensione senza introdurre un classificatore di parole nel codice. Descrivete quali candidati il programma possa rendere disponibili, quali metadati siano necessari, quale scelta competa al modello e quali controlli vadano ripetuti prima del recupero. Aggiungete una condizione nella quale il sistema debba chiedere aiuto. La soluzione deve anche impedire che un candidato di un'altra conversazione diventi accessibile soltanto perché il suo titolo sembra pertinente.

La lezione del cambiamento non è che basti spostare una condizione dentro un prompt. Occorre assegnare correttamente le responsabilità e collegarle a prove diverse. Nel prossimo capitolo costruiremo un laboratorio Swift che applica lo stesso metodo a un controllo più piccolo: verificare l'identità delle fonti e l'esistenza dei passaggi citati, mostrando subito ciò che quel controllo non può dimostrare.

Riferimenti per proseguire

La guida ai sorgenti identifica revisioni ed estratti. Il metodo è approfondito in *Dal prompt al software*, nei capitoli sul contesto, sui criteri di accettazione e sulla responsabilità del codice.

4. Preparare un'esperienza ripetibile

Nei capitoli precedenti abbiamo incontrato una distinzione che deve diventare concreta: verificare il programma e valutare il modello sono lavori diversi. Possiamo imparare a seguirla senza collegare subito un servizio esterno. Ci serve un caso piccolo nel quale ingressi, regole e risultato atteso siano comprensibili e nel quale il limite del controllo sia visibile.

Prepariamo quindi un laboratorio Swift sulle fonti del confronto. Il programma riceve documenti e citazioni già scelti e controlla identità, revisioni ed esistenza degli estratti. Non interpreta l'incarico e non sceglie uno strumento. È un componente di verifica didattico, costruito per questo libro. Il suo scopo è farci leggere una proprietà prima di ritrovarla fra le molte responsabilità di Velia.

Il caso più importante contiene una frase sbagliata accompagnata da citazioni formalmente corrette. Il programma deve accettare i riferimenti senza per questo certificare la frase. Se alla fine sappiamo spiegare perché, abbiamo acquisito un criterio che servirà per tools, memoria, documenti e consegna.

4.1 Isolare una domanda che possiamo controllare

Riprendiamo le fonti A e B. La prima dice che la consegna avviene entro dieci giorni dalla conferma dell'ordine; la seconda indica quindici giorni con lo stesso riferimento. Il rapporto preparato dichiara la differenza e cita entrambe le frasi. I documenti e il rapporto sono inventati per l'esempio e conservati come ingressi curati.

La domanda del programma è limitata: le fonti attese sono disponibili nella revisione indicata e gli estratti citati compaiono nei testi corrispondenti? Non chiediamo ancora se il confronto sia utile o completo per tutto l'incarico. La parte relativa all'assistenza rimane nel caso esteso del libro e non viene qualificata da questo laboratorio.

Questo confine permette di assegnare risultati attesi chiari. Se B manca, il confronto non dispone di tutte le fonti richieste. Se l'etichetta di revisione cambia, la citazione precedente non può essere accettata come riferimento alla versione corrente. Se il passaggio non esiste, il riferimento non è valido. Se il testo della conclusione è sbagliato ma i passaggi esistono, il controllo non possiede una regola per riconoscerne il significato.

La figura 4.1 mostra la separazione. Il laboratorio segue il ramo dei controlli sui dati. La valutazione del significato resta una responsabilità successiva, che può richiedere una persona competente o un confronto progettato con un modello reale.

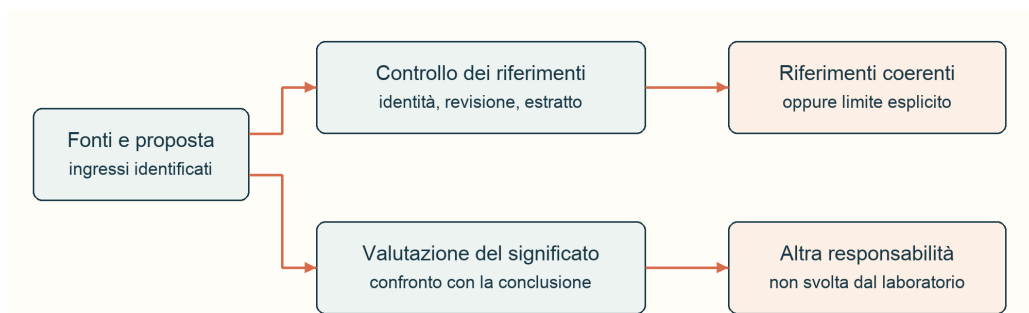


Figura 4.1 - Un riferimento può essere rifiutato attraverso identità, revisione o estratto. Superare questi controlli rende il riferimento utilizzabile, ma lascia aperta la domanda se sostenga la conclusione.

4.2 Che cosa serve e che cosa viene conservato

Il laboratorio usa un compilatore Swift 6 e Python 3 disponibili nel sistema. Swift esegue il controllo; Python prepara la compilazione e i casi senza creare un ambiente permanente. Non servono credenziali o un account del modello. Non vengono letti archivi o conversazioni dell'applicazione Velia.

Per riconoscere gli strumenti disponibili potete eseguire `swift --version` e `python3 --version`. La versione del compilatore riguarda la riproduzione del programma; non prova la compatibilità di tutta l'app macOS o delle sue dipendenze. Il laboratorio è deliberatamente separato dalla build completa del prodotto.

I materiali sono nella cartella del capitolo 4. `Sources/VerificaFonti.swift` contiene il programma. La cartella `fixtures` contiene i cinque casi JSON. `esegui.py` è il driver, cioè il programma che prepara ambiente, compilazione ed esecuzione. Le istruzioni locali descrivono il contratto prima del codice.

Un ingresso curato, o fixture, è un caso scelto per porre una domanda precisa al programma. Qui comprende fonti attese, testi e affermazioni con citazioni. Rimane nel progetto perché permette di ripetere la domanda. L'eseguibile compilato, le cache e la diagnostica prodotta da un tentativo sono invece intermedi: servono durante la verifica e vengono rimossi alla fine del ciclo.

La distinzione è operativa. Non vogliamo un laboratorio che funzioni soltanto grazie a un file generato nella prova precedente. Ogni esecuzione deve poter ricostruire ciò che

serve a partire dai sorgenti e dagli ingressi. Non vogliamo neppure che la pulizia dei test elimini documenti o dati dell'applicazione: il driver lavora in una directory privata e non tocca quelle risorse.

Buona pratica - La prova deve avere un ambiente riconoscibile

Per ripetere un controllo servono il sorgente, i dati, le versioni degli strumenti e il criterio previsto. Una copia del messaggio «tutto bene» non consente di ricostruire la domanda. Un risultato non riuscito va letto prima della pulizia, mentre i materiali che permettono di ripetere il caso rimangono disponibili.

4.3 Leggere i dati prima del controllo

Il file JSON ha tre gruppi principali: `fontiAttese`, `fonti` e `affermazioni`. Le fonti attese descrivono il perimetro del caso; le fonti disponibili portano identità, revisione e testo; le affermazioni collegano una frase alle citazioni che la accompagnano.

Questa è una delle identità attese:

```
{
  "id": "B",
  "revisione": "r1"
}
```

L'identificatore permette di distinguere B da A. La revisione permette di non trattare automaticamente due contenuti come la stessa fonte. In questo laboratorio `r1` è un'etichetta preparata negli ingressi: non è una firma e non deriva da un controllo crittografico dei byte. La fiducia nella preparazione della fixture è una condizione dell'esperimento. Nei sistemi reali occorre stabilire come l'identità venga attribuita e collegata al contenuto.

Una citazione contiene l'identità della fonte, la sua revisione e l'estratto. Questo frammento corrisponde al documento B del caso con tutti gli ingressi:

```
{
  "fonte": "B",
  "revisione": "r1",
  "estratto": "La consegna avviene entro quindici giorni dalla conferma dell'ordine."
}
```

La struttura rende visibili tre possibili errori. L'identificatore può non esistere, la revisione può non coincidere e l'estratto può non comparire nel testo. Il programma li può controllare senza interpretare il significato della frase della persona. È lo stesso confine incontrato nel capitolo 3: un confronto di identità o di contenuto non diventa per questo un classificatore dell'intento.

Le affermazioni mantengono separato il testo dalle citazioni. Questa separazione è utile proprio perché la presenza delle citazioni non dimostra la verità del testo. Se avessimo un unico campo con una risposta libera e un elenco indistinto di fonti, sarebbe più difficile descrivere quale collegamento vogliamo controllare.

4.4 I tipi Swift traducono una responsabilità

Prima di leggere il controllo osserviamo i tipi che rappresentano gli ingressi. Il programma completo è nel file sorgente; qui selezioniamo due strutture:

```
struct Citazione: Decodable {
    let fonte: String
    let revisione: String
    let estratto: String
}
struct Affermazione: Decodable {
    let testo: String
    let citazioni: [Citazione]
}
```

Una struct raccoglie valori che appartengono allo stesso concetto. `let` indica una proprietà che, una volta inizializzata, non viene riassegnata attraverso quel valore. `String` è il tipo usato per il testo; `[Citazione]` rappresenta una collezione di citazioni. `Decodable` dichiara che il tipo può essere ricostruito da una rappresentazione esterna attraverso i meccanismi di decodifica adottati.

Queste definizioni non provano che l'estratto sia corretto. Consentono al programma di leggere campi distinti, sui quali applicherà i controlli. Un ingresso che non può essere decodificato viene rifiutato prima della verifica dei riferimenti. Il semplice fatto di essere JSON valido, però, non lo rende un caso coerente.

Il programma controlla anche che gli identificatori non siano ripetuti, che i campi necessari non siano vuoti e che le fonti disponibili appartengano all'insieme previsto. Prima di creare un dizionario, cioè una struttura che associa una chiave a un valore, verifica l'unicità delle chiavi. Così due fonti con lo stesso identificatore non vengono accettate come una situazione normale.

Questo è un piccolo ponte a Swift. Non serve conoscere tutta la lingua per leggere la responsabilità delle strutture. Nei capitoli successivi svilupperemo protocolli, concorrenza e gestione dello stato quando entreranno nell'architettura di Velia. Il riferimento generale è *The Swift Programming Language*, pubblicato dal progetto Swift; qui usiamo soltanto i costrutti necessari al caso.

4.5 Il punto nel quale una citazione diventa controllabile

Il controllo prende una citazione e cerca la fonte corrispondente nel dizionario. Se la fonte manca ma era attesa, conserva l'assenza. Se l'identificatore non appartiene al caso, lo segnala come riferimento non valido. Quando la fonte esiste, applica questa condizione, estratta dal programma completo:

```
guard citazione.revisione == fonte.revisione,
    presente(citazione.estratto),
    fonte.testo.contains(citazione.estratto)
else {
    let numero = indice + 1
    let origine = citazione.fonte
    print("citazione_non_valida: affermazione \%(numero), fonte \%(origine)")
    riferimentoNonValido = true
    continue
}
```

Le condizioni devono essere tutte vere. La prima confronta le etichette di revisione. La seconda richiede un estratto non vuoto secondo la funzione `presente`. La terza cerca l'estratto nel testo della fonte. Se una condizione manca, il programma registra il problema del controllo e passa alla citazione successiva. Il risultato finale conserva l'errore anche quando altri riferimenti sono corretti.

Il confronto del testo riguarda la presenza dell'estratto come sottostringa secondo il comportamento delle stringhe Swift. Non riconosce parafrasi, non valuta la qualità di una traduzione e non dimostra che la frase citata esaurisca le condizioni del documento. Sono limiti da mantenere vicini al controllo, perché cambiano il significato del suo esito.

Il nome `riferimentoNonValido` descrive il problema rilevato. Sarebbe fuorviante chiamarlo `rispostaFalsa`, perché il programma non ha analizzato tutta la risposta. Lo stesso vale per il percorso positivo: il messaggio previsto è `riferimenti_coerenti`; `significato_non_valutato`. È una parte del contratto del sorgente, non un'affermazione di correttezza generale.

4.6 Eseguire i casi senza lasciare risultati nel progetto

I comandi seguenti si eseguono dalla radice del repository del libro. Il gestore `gate.py` prepara il ciclo di controllo e ne governa la pulizia. Il driver Python compila il sorgente in uno spazio privato, dirige lì la cache dei moduli del compilatore ed esegue i casi. Il controllo finale verifica che non rimangano risultati da conservare.

```
python3 quality-gates/storage/gate.py run -- \
python3 ai-agentica/esempi/velia/capitolo-04/esegui.py \
--verifica
python3 quality-gates/storage/gate.py check
```

--verifica confronta gli esiti dei cinque casi con i criteri preparati. Il comando non chiama un modello. Il suo buon esito significa che il programma si comporta secondo quei criteri nel laboratorio, compreso il limite che vogliamo osservare. Non dimostra il funzionamento di un agente né la qualifica del codice completo di Velia.

Il compilatore produce un eseguibile a partire dal sorgente; la cache conserva intermedi che possono accelerare la compilazione. Qui entrambi appartengono al ciclo e vengono rimossi. Il driver usa un gestore di contesto Python per la pulizia ordinaria, mentre il gestore esterno controlla il ciclo anche in caso di interruzione. Non vengono create installazioni permanenti o copie dei dati dell'app.

Se Swift non è disponibile, il driver segnala il requisito mancante. Se il sorgente non compila, non si passa ai casi come se il controllo fosse riuscito. Se un caso restituisce un codice diverso da quello previsto, il driver termina con errore. La pulizia non trasforma un errore della prova in un risultato favorevole.

Per osservare un singolo caso possiamo usare:

```
python3 quality-gates/storage/gate.py run -- \
python3 ai-agentica/esempi/velia/capitolo-04/esegui.py \
--caso manca-b
```

Il caso della fonte mancante deve terminare con il codice previsto per quell'assenza. Un'uscita diversa da zero è qui l'effetto che stiamo studiando; non dobbiamo modificarla soltanto per ottenere un comando verde. Il criterio è riconoscere quale domanda sia stata posta e quale risultato debba seguirne.

4.7 Cinque casi, cinque letture del risultato

Il caso completo contiene le due fonti attese e una frase che descrive correttamente il passaggio da dieci a quindici giorni. Il codice 0 indica che i riferimenti controllati sono coerenti. Il nome del caso riguarda la disponibilità degli ingressi di questo esperimento, non la completezza del confronto dell'assistenza o di tutto l'incarico esteso.

Nel caso manca-b la fonte B è assente. La citazione che la richiama non è verificabile; il programma conserva quel limite e restituisce il codice 2. Non usa il testo dell'affermazione per ricostruire la fonte mancante. Anche una frase che per caso fosse vera rimarrebbe priva del sostegno richiesto dal caso.

Nel caso *revisione-superata* la fonte B disponibile porta r2, mentre la richiesta e la citazione si riferiscono a r1. Il codice 3 segnala la mancata corrispondenza. Non abbiamo ancora stabilito se il contenuto sia cambiato in modo importante: il controllo dice che il riferimento non identifica la revisione attesa. Per decidere il significato del cambiamento occorre leggere i dati pertinenti.

Nel caso *citazione-inesistente* l'estratto preparato parla di due giorni, ma non compare nella fonte B. Il controllo lo rifiuta con codice 3. Questo è un errore che possiamo rilevare senza un giudizio sulla qualità della prosa. Nel caso *frase-non-sostenuta*, invece, lasciamo invariate le citazioni e scriviamo che il termine si riduce da dieci a cinque giorni. Gli estratti esistono ancora e le revisioni coincidono: il controllo restituisce 0.

Quest'ultimo esito è il punto centrale del laboratorio. Il programma non ha letto il significato della conclusione. I suoi controlli rimangono utili, perché intercettano altri errori, ma non possono essere presentati come ciò che non verificano. Una revisione semantica dovrebbe confrontare «si riduce a cinque» con il passaggio della fonte che dice quindici. L'esistenza della citazione è una condizione utile a quella revisione, non il suo sostituto.

Codice	Significato nel laboratorio
0	Nessun problema nei riferimenti controllati; significato non valutato.
2	Almeno una fonte attesa manca e non sono stati rilevati errori di riferimento con priorità maggiore.
3	Almeno un riferimento o una revisione non rispetta il contratto.
4	I dati o i requisiti del programma non consentono il controllo.

Se nello stesso caso esistono sia una fonte mancante sia un riferimento non valido, il programma restituisce 3. La scelta della priorità è parte del contratto di questo piccolo eseguibile. Le segnalazioni durante il ciclo permettono comunque di osservare più problemi; non attribuiamo al solo codice finale una descrizione completa della situazione.

4.8 Dal laboratorio al progetto reale

In Velia le fonti non arrivano da una fixture preparata insieme al rapporto. Vengono acquisite attraverso strumenti, permessi, inventari e letture che possono interrompersi. I riferimenti devono rimanere collegati alle risorse e ai contenuti effettivi. L'architettura completa deve quindi affrontare problemi che il laboratorio mantiene fuori dal proprio confine.

Il nostro programma si fida delle etichette di revisione contenute negli ingressi. Se qualcuno cambia il testo e mantiene `r1`, non dimostra che i byte siano quelli della versione originaria. Un collegamento più forte può richiedere un'impronta del contenuto, cioè un valore calcolato dai byte, e una provenienza controllata di quell'impronta. Anche l'impronta, da sola, non stabilisce chi abbia prodotto il documento o se il contenuto sia vero.

Il laboratorio richiede che un'affermazione abbia almeno una citazione, ma non dimostra che ogni fonte sia usata correttamente o che tutti i punti della richiesta siano coperti. Una dichiarazione può avere due riferimenti corretti e omettere l'assistenza. Quando progetteremo i criteri dell'incarico dovremo mantenere distinta la copertura dei materiali dalla copertura delle domande e dal sostegno delle conclusioni.

Il controllo lavora inoltre su un testo già disponibile. Non valuta l'estrazione di un PDF, la corretta lettura di una tabella, una traduzione o una citazione che dipenda dalla pagina precedente. Queste responsabilità entreranno nei capitoli sui documenti. Isolare un confine serve a capire come verificarlo e a sapere quando occorra un altro controllo.

4.9 Una prova di trasferimento

Progettate una variante con tre documenti: A, B e un allegato C che cambia le condizioni di assistenza. Descrivete quali identità e revisioni devono essere attese e quale differenza vada introdotta nei dati. Potete aggiungere una fixture, senza modificare il significato degli esiti del programma. Prima di eseguirla scrivete il risultato atteso e il limite che rimane.

Progettate poi un caso che superi tutti i controlli ma resti inadatto alla consegna. Per esempio, le citazioni possono essere corrette e il testo può tacere una condizione decisiva contenuta altrove. Indicate quale controllo aggiuntivo servirebbe e quale ingresso gli permetterebbe di lavorare. Non basta chiamarlo «revisore»: occorre dire che cosa deve confrontare e come riconoscere un problema.

Le soluzioni ragionate degli apparati mostrano possibili risposte. Il criterio da trasferire è già disponibile: una verifica affidabile dichiara la propria domanda, usa ingressi identificabili e mantiene visibile ciò che resta fuori. Il capitolo successivo entrerà nei componenti Swift di Velia con questo stesso metodo. Il piccolo controllo delle fonti sarà uno strumento di comprensione; il progetto continuerà a comprendere molte altre responsabilità.

5. Dal progetto ai componenti

Il verificatore del capitolo precedente ci ha permesso di osservare una proprietà piccola: un riferimento può essere coerente senza che la conclusione sia sostenuta. Il programma riceveva tutti i dati da un ingresso preparato. In Velia quei dati devono arrivare attraverso un'applicazione: una persona scrive, un coordinatore prepara il lavoro, un trasporto comunica con il modello, gli strumenti acquisiscono informazioni e un deposito conserva ciò che è successo.

Adesso vogliamo orientarci nel progetto senza ridurlo al programma del laboratorio. La domanda è concreta: se cambiamo il modo di ottenere un dato, quali parti devono cambiare e quali controlli devono continuare a valere? La risposta richiede due mappe. Una descrive che cosa viene compilato insieme; l'altra segue le responsabilità durante il lavoro. Confonderle porta a cercare una funzione nel posto sbagliato o ad attribuire a un modulo una garanzia che il suo nome non dimostra.

Leggeremo il pacchetto reale di Velia, nella stessa revisione adottata nei capitoli precedenti. Poi useremo un secondo laboratorio Swift per sostituire una lettura in memoria con una lettura da file. È un esperimento sul confine fra componenti, distinto dal trasporto del modello e dalla costruzione dell'applicazione completa.

5.1 Una cartella di sorgenti non è ancora una mappa del sistema

Apriamo un repository troviamo file, cartelle, risorse e strumenti di sviluppo. Questa disposizione aiuta a navigare, ma non basta a capire le dipendenze. Due classi in cartelle diverse possono essere compilate nello stesso modulo; due file vicini possono appartenere a prodotti diversi. Un nome come «Core» indica una scelta del progetto, non una proprietà universale.

Cominciamo quindi da `Package.swift`. È il manifesto letto da Swift Package Manager, lo strumento che organizza costruzione e dipendenze dei pacchetti Swift. Il manifesto è scritto usando il linguaggio Swift, ma descrive come costruire il programma: non è il punto nel quale l'assistente decide che cosa fare con una richiesta.

Nel lessico del manifesto, un *pacchetto* raccoglie sorgenti e configurazione. Un *target* è un'unità di costruzione con i propri sorgenti e le dipendenze dichiarate. Per un target Swift del nostro percorso, il codice viene organizzato in un modulo che gli altri possono importare. Un *prodotto* è invece ciò che il pacchetto rende disponibile

all'esterno, come un eseguibile o una libreria. Uno stesso pacchetto può costruire più prodotti; non è necessario esporre come prodotto separato ogni suo target interno.

Questa distinzione spiega un fatto altrimenti sorprendente: Velia usa VeliaCore nel proprio desktop, anche se il manifesto non dichiara un prodotto libreria chiamato VeliaCore. Il target esiste e può essere una dipendenza interna. L'elenco dei prodotti non è l'inventario di tutti i componenti.

Riferimento essenziale - Leggere il manifesto

La **documentazione di PackageDescription**, pubblicata dal progetto Swift, descrive pacchetti, target, prodotti e requisiti delle dipendenze. Qui la usiamo per interpretare il manifesto di Velia; le responsabilità dell'assistente devono invece essere ricostruite dai suoi sorgenti. La validità del manifesto non dimostra il comportamento dell'applicazione.

5.2 Che cosa dichiara il pacchetto di Velia

Il manifesto della revisione del libro comincia con `// swift-tools-version: 6.0` e indica macOS 15 come piattaforma minima. La prima dichiarazione stabilisce il livello richiesto degli strumenti e del manifesto; la seconda riguarda il sistema sul quale il prodotto è destinato a funzionare. Non significa che Swift 6 richieda sempre macOS 15 o che ogni programma Swift dipenda dalle stesse interfacce di sistema usate da Velia.

Il pacchetto dichiara sette target e cinque prodotti. Tre prodotti sono eseguibili: desktop, host del Terminale e host Python. In questo progetto gli host sono programmi ausiliari che supervisionano attività avviate dall'applicazione; ne leggeremo il ruolo fra poco. Gli altri due prodotti sono librerie dinamiche per acquisizione web e interfaccia del Terminale. Una libreria dinamica viene collegata e caricata dal programma che la usa; non diventa per questo un servizio autonomo o un agente distinto.

Per leggere la relazione centrale ci basta questo estratto reale del manifesto, riformattato senza cambiare i valori:

```
.executableTarget(  
    name: "VeliaDesktop",  
    dependencies: ["VeliaCore", "VeliaCrawling", "VeliaTerminalUI"],  
    path: "Sources/VeliaDesktop",  
    exclude: ["Resources"]  
)
```

Il desktop dipende dai tre target indicati. `path` identifica la cartella dei suoi sorgenti; `exclude` esclude Resources dalla gestione di quel target nel manifesto. Quest'ultima voce non permette di concludere che la distribuzione dell'app non abbia risorse: per ricostruire il pacchetto installabile occorre leggere anche gli strumenti che lo assemblano. La compilazione e il collegamento del codice, che qui chiameremo `build`, preparano eseguibili e librerie. La preparazione dell'app da consegnare deve occuparsi anche delle altre risorse necessarie.

La figura 5.1 rende visibili le dipendenze dichiarate. Seguiamo prima le frecce che partono dal desktop, poi quelle dell'acquisizione web e del Terminale. Gli host in basso sono altri eseguibili; il manifesto non li elenca come dipendenze di compilazione del desktop. Il loro impiego a esecuzione avviata richiede un altro punto di lettura.

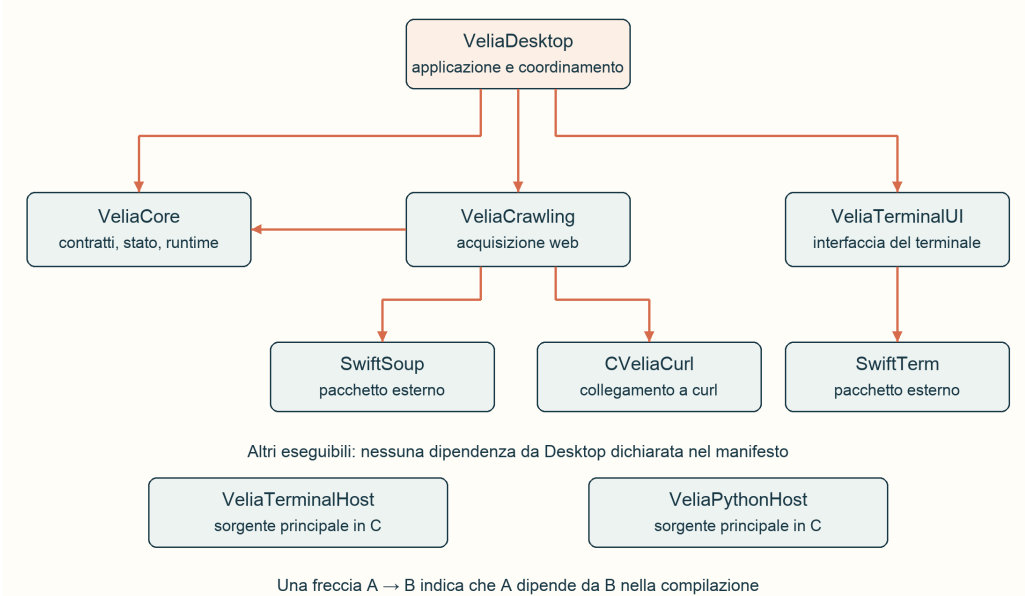


Figura 5.1 - La freccia va dal componente che dipende a quello usato. Il disegno descrive il manifesto della revisione adottata, non tutti gli scambi durante l'esecuzione. Gli eseguibili ausiliari non sono agenti aggiuntivi.

Nel manifesto compaiono anche due pacchetti esterni, con versioni richieste esatte: SwiftSoup 2.13.5 e SwiftTerm 1.20.0. Il primo è una dipendenza di `VeliaCrawling`, il secondo di `VeliaTerminalUI`. `CVeliaCurl` dichiara inoltre il collegamento alla libreria `curl`. Sono dati della revisione scelta, non una raccomandazione sulle versioni più recenti. Le interfacce di programmazione del sistema operativo, dette API, e i requisiti dell'ambiente rimangono necessari anche quando non compaiono nell'elenco dei pacchetti esterni.

5.3 Seguire un incarico attraverso le responsabilità

Torniamo al confronto fra i documenti A e B. Il messaggio della persona entra nell'applicazione desktop; il codice deve conservare il rapporto fra richiesta, conversazione ed esecuzione. Quando serve una decisione del modello, il coordinatore prepara le informazioni e usa il trasporto. Se la risposta propone un tool, il percorso deve controllare ed eseguire la capacità ammessa, registrando l'esito prima di proseguire.

Questa è una mappa di responsabilità. Per ritrovarla nel codice ci servono pochi punti di ingresso, ciascuno con una domanda da seguire.

Punto di lettura	Domanda alla quale risponde
AssistantController, nel desktop	Come vengono gestiti richiesta e stato presentato alla persona?
ChatAgentCoordinator, nel desktop	Come si prepara il turno e si collega una decisione al seguito del lavoro?
ChatModelTransport, nel desktop	Quale contratto usa il coordinatore per ottenere una risposta del modello?
AgentRuntime, nel nucleo	Come una capacità registrata viene controllata, eseguita e conclusa?
AgentStore, nel nucleo	Come si leggono e aggiornano esecuzioni, passi e dati operativi?

Non dobbiamo memorizzare i nomi per comprendere il percorso. Ci servono per poter verificare una frase come «il programma registra il risultato»: chi lo riceve, chi lo controlla e quale componente conserva il record? La guida ai sorgenti indica i file della revisione descritta.

Una conseguenza della lettura è particolarmente utile. Il ciclo della chat e il trasporto del modello non sono definiti nel target `VeliaCore`: in questa revisione appartengono al desktop. Il nucleo contiene contratti, stato e runtime di esecuzione degli strumenti. Separare queste responsabilità nella spiegazione non autorizza a ridisegnare il repository come se fossero già moduli diversi.

Questo chiarisce anche il termine *runtime* incontrato nell'introduzione. Lo abbiamo usato per la responsabilità generale di governare un'esecuzione. La classe concreta `AgentRuntime` svolge una parte di quel lavoro: gestisce strumenti registrati e collabora con il deposito. Non contiene da sola tutte le decisioni della chat. Un termine

architetturale e un identificatore del codice possono avere ampiezze differenti; seguirne gli usi evita di attribuire alla classe l'intero sistema.

5.4 Un contratto permette di cambiare il collegamento

Se il coordinatore costruisse direttamente ogni dettaglio della connessione, sarebbe difficile provarlo senza il servizio esterno. Nel codice troviamo invece una proprietà dichiarata così:

```
let transport: any ChatModelTransport
```

`ChatModelTransport` è un protocollo Swift: dichiara le operazioni richieste a un componente, lasciando alle implementazioni il modo di realizzarle. `any` indica qui un valore la cui implementazione concreta può variare, purché rispetti quel protocollo. `let` impedisce di riassegnare la proprietà dopo l'inizializzazione; non dimostra che ogni oggetto raggiungibile attraverso di essa sia immutabile.

Il costruttore del coordinatore riceve il trasporto e, se non ne viene fornito uno, usa `OpenAIChatModelTransport`. Questo modo di ricevere un collaboratore dall'esterno viene chiamato *iniezione delle dipendenze*. Il nome descrive una scelta semplice: chi assembla il sistema può fornire un componente conforme al contratto, senza riscrivere il coordinatore per ogni implementazione.

Il protocollo reale comprende completamento e conteggio dei token di ingresso. Nel metodo `complete` riceve il corpo della richiesta come `Data`, cioè una sequenza di byte, insieme a credenziale, modalità di autenticazione e scadenza. Riceve anche una funzione da richiamare per consegnare parti del testo. Restituisce i dati della risposta oppure produce un errore. Non mostriamo ancora l'intera firma: attesa asincrona e consegna progressiva saranno sviluppate nei capitoli 6 e 7, prima di usarle per leggere il percorso completo.

La separazione offre un punto nel quale inserire un trasporto controllato per una prova. Non rende intercambiabile qualsiasi servizio cambiando soltanto un indirizzo. Un altro provider può avere formati, autenticazione, eventi ed errori differenti. Occorre adattarli al contratto e verificare che il significato degli esiti rimanga quello atteso dal chiamante. La sostituibilità riguarda obblighi precisi, non il solo fatto che due metodi restituiscano byte.

Approfondimento - Compilare non significa rispettare tutto il contratto

Il compilatore può verificare la presenza di un metodo con i tipi richiesti. Non dimostra che il metodo rispetti una scadenza o restituisca il documento domandato. Alcuni obblighi sono espressi dai tipi, altri dalle condizioni del programma e dalle prove. Nei prossimi paragrafi costruiremo proprio un trasporto formalmente corretto che restituisce una revisione da rifiutare.

5.5 Il deposito possiede dati che il modello non vede automaticamente

Nel nucleo, AgentStore è dichiarato come actor, un costrutto Swift per proteggere l'accesso al proprio stato nel lavoro concorrente. Il capitolo 6 spiegherà attese e accessi: per ora ci basta distinguere questa protezione dalla persistenza. Un actor può contenere soltanto dati volatili; la parola non implica che qualcosa venga salvato sul disco.

La persistenza del deposito di Velia si riconosce seguendo il collegamento con AgentDatabase, che usa SQLite, un motore di database locale. I metodi del deposito leggono e aggiornano progetti, esecuzioni, passi e altri record. Conversazioni e memoria hanno poi strutture e percorsi propri. Non tutta l'informazione dell'assistente si riduce a un unico elenco di messaggi.

Se la richiesta corrente riguarda A e B, un record salvato su C non entra automaticamente nel contesto. Il coordinatore deve chiedere i dati necessari e prepararli per il modello, con accessi e ruoli adeguati. È lo stesso limite osservato nel recupero del capitolo 3: avere conservato un tentativo non decide che sia pertinente al nuovo incarico.

Anche qui il confine non è soltanto organizzativo. Il trasporto del modello non deve diventare il proprietario implicito dello stato del compito. Se una connessione si interrompe, la sua interruzione non può essere interpretata come cancellazione di una lettura già registrata o come prova che un effetto esterno non sia avvenuto. Per ricostruire il lavoro dobbiamo tornare ai componenti che possiedono quei fatti.

5.6 Un processo ausiliario introduce un altro tipo di confine

I target degli host ci obbligano a distinguere un ulteriore concetto. Un modulo è un confine del codice e dei nomi disponibili. Un processo è un programma in esecuzione, con una propria vita che può terminare separatamente da quella dell'applicazione che lo ha avviato. Usare un modulo non equivale ad avviare un nuovo processo.

Nel pacchetto esaminato i sorgenti principali di `VeliaTerminalHost` e `VeliaPythonHost` sono scritti in C. Il primo supervisiona il terminale e i processi collegati; il secondo supervisiona il percorso di esecuzione Python tramite Docker. Il nome dell'host Python non indica quindi che il suo supervisore sia uno script Python. L'applicazione può usare Swift nel percorso principale e includere componenti di altri linguaggi quando il lavoro lo richiede.

La separazione introduce domande concrete: chi avvia il processo, con quali argomenti, chi ne osserva la conclusione e che cosa rimane incerto se il supervisore si arresta? Le affronteremo nei capitoli dedicati. La presenza di un processo o di un contenitore non dimostra da sola l'isolamento necessario: servono controlli su risorse, identità, comunicazione ed effetti effettivamente consentiti.

Il criterio per leggere la figura 5.1 è ora più preciso. Una freccia del manifesto ci informa della compilazione. Per spiegare l'esecuzione di un host dobbiamo seguire il codice che lo avvia e il protocollo con cui comunica. Il primo disegno non può sostituire il secondo percorso.

5.7 Costruire un confine piccolo, ma compilabile

Passiamo a un esperimento nel quale possiamo cambiare un collegamento senza intervenire su Jarvis. Il laboratorio del capitolo 5 è un pacchetto con tre target. `ConfrontoContratti` dichiara fonte e protocollo; `ConfrontoNucleo` controlla ciò che arriva; `ConfrontoCLI` è l'applicazione da riga di comando che sceglie e collega i componenti.

La responsabilità viene dal capitolo 4, ma la domanda cambia. Prima ricevevamo un caso JSON già completo e controllavamo anche le citazioni. Ora acquisiamo una fonte attraverso un collaboratore e verifichiamo soltanto identità e revisione. Il programma non è un nuovo agente, non comprende la richiesta e non sostituisce il verificatore precedente.

Il contratto di lettura è breve:

```
public protocol TrasportoFonti {  
    func leggi(id: String) throws -> Fonte  
}
```

Fonte conserva id, revisione e testo. Il protocollo richiede una lettura dato un identificatore. throws ammette un errore, per esempio se la fonte manca. public rende disponibile la dichiarazione agli altri moduli; senza una visibilità adatta, separare i file in target diversi non basta per poter usare i tipi attraverso il confine.

Il nucleo importa soltanto ConfrontoContratti. Non conosce il tipo che legge i file né quello che conserva una mappa in memoria. Il suo controllo completo è questo:

```
public func acquisisci(  
    id: String, revisione: String,  
    tramite trasporto: any TrasportoFonti  
) throws -> Fonte {  
    let fonte = try trasporto.leggi(id: id)  
    guard fonte.id == id else { throw ErroreFonte.identita }  
    guard fonte.revisione == revisione else { throw ErroreFonte.revisione }  
    return fonte  
}
```

In Swift tramite è l'etichetta usata nel punto di chiamata, mentre trasporto è il nome impiegato nel corpo. try segnala una chiamata dalla quale può arrivare un errore. Prima otteniamo una fonte; poi controlliamo che sia quella richiesta e nella revisione prevista. Il risultato viene restituito soltanto dopo entrambe le verifiche.

Se avessimo affidato il controllo soltanto al trasporto da file, una seconda implementazione potrebbe saltarlo. Qui i due controlli appartengono al nucleo che usa il dato. Il protocollo non può impedire a un'implementazione di restituire C quando abbiamo chiesto B; il nucleo può però riconoscere la violazione prima di accettarne il risultato.

5.8 La modifica: dalla memoria a un file

La prima implementazione, TrasportoInMemoria, cerca l'identificatore in una mappa preparata. La seconda, TrasportoDaFile, usa una mappa che associa A e B a percorsi JSON prestabiliti, legge i byte e li decodifica come Fonte. I due trasporti appartengono all'applicazione; il nucleo dipende dal loro protocollo, non dai tipi concreti.

La modifica circoscritta avviene nel punto in cui assembliamo il programma: scegliamo la lettura da file al posto della mappa in memoria. Il nucleo e il suo contratto rimangono identici. Nel laboratorio entrambe le implementazioni sono già

presenti nello stesso eseguibile, così possiamo confrontarle senza ricompilare una seconda versione del controllo.

Il caso positivo richiede A e B nella revisione r1. Prepariamo poi B con r2, lasciando invariato il testo: il rifiuto deve dipendere dalla revisione, senza dedurre che il contenuto sia necessariamente diverso. Un altro caso restituisce una fonte con identità C alla richiesta di B. Infine togliamo B. Nel trasporto da file manca il file curato; in quello in memoria manca la voce della mappa. La causa tecnica cambia, il significato applicativo dell'assenza resta riconoscibile.

Scenario	Esito atteso per entrambi i trasporti	Che cosa controlliamo
Fonti correnti	Codice 0 e identità A e B con r1.	Il nucleo accetta le due fonti richieste.
B nella revisione r2	Codice 3, revisione non corrispondente.	Il cambio di collegamento non salta il controllo della revisione.
C restituito al posto di B	Codice 3, identità non corrispondente.	La conformità al protocollo non basta a far accettare il dato.
B assente	Codice 2, fonte assente.	La mancanza non diventa un testo vuoto trattato come lettura riuscita.

Un esito positivo non conferma il significato dei documenti. Le revisioni sono etichette curate, come nel capitolo 4; non provano l'integrità dei byte. Non vengono controllate citazioni o completezza della sintesi. La proprietà osservabile è che entrambi i percorsi attraversano il medesimo controllo di acquisizione.

5.9 Ripetere il confronto fra i componenti

Servono Swift 6 con Swift Package Manager e Python 3. Dalla radice del repository eseguiamo:

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-05/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

Il driver prepara una copia privata del manifesto e dei sorgenti, compila una sola volta e usa l'eseguibile per gli otto casi: quattro scenari per due trasporti. Per ogni caso confronta codice di uscita e messaggio con i criteri preparati. Nel caso positivo

controlla anche le identità e revisioni stampate. Un errore di compilazione impedisce di passare ai casi come se i moduli fossero disponibili.

Build, cache e configurazioni del gestore Swift appartengono al ciclo privato e vengono rimossi. Nel repository rimangono i sorgenti e le fixture curate, non le risposte dell'ultima prova. Non servono rete, credenziali, l'archivio operativo o l'installazione di Velia. Questo esperimento verifica il pacchetto didattico; non dichiara compilato o qualificato il prodotto reale.

Per osservare soltanto il rifiuto della revisione superata possiamo scegliere `--trasporto file --scenario revisione-superata` al posto di `--verifica`. Il singolo programma restituisce 3, il rifiuto previsto. Il driver collettivo restituisce invece 0 quando tutti i casi rispettano le attese, compresi i rifiuti. Sono due domande diverse: che cosa accade in questo caso e se l'insieme dei casi segue i criteri.

5.10 Una variante che mette alla prova il contratto

Immaginate di sostituire i file locali con un servizio documentale remoto. Possiamo riusare l'obiettivo: ottenere una fonte identificata e rifiutare una revisione diversa. Non possiamo concludere che il protocollo attuale basti per una connessione che può restare in attesa. La funzione didattica è sincrona: non esprime ancora attesa asincrona, cancellazione o scadenza. Un adattatore che non restituisce il controllo impedisce anche al nucleo di esaminare il dato.

L'esercizio chiede di descrivere l'estensione prima di scrivere una chiamata di rete. Indicate quali informazioni devono attraversare il confine, dove mantenere il controllo di identità e revisione e come distinguere fonte assente, accesso negato e servizio non raggiungibile. Aggiungete un caso nel quale la persona annulla mentre la lettura è in corso. Quale componente deve ricevere l'annullamento e che cosa deve accadere a una risposta arrivata più tardi?

Proponete poi una modifica più semplice: mostrare ciascuna fonte su una riga separata, con identificatore e revisione. Motivate perché riguardi la presentazione dell'applicazione e non richieda di spostare il controllo di revisione nel trasporto. Le soluzioni degli apparati discutono entrambe le varianti, senza anticipare un'implementazione remota che non abbiamo ancora costruito.

Il criterio che possiamo già usare è cercare il proprietario di una responsabilità e le informazioni necessarie per verificarla. Separare i moduli rende visibili alcune dipendenze; conservare un contratto permette una sostituzione controllabile. Per capire che cosa accada quando due operazioni si sovrappongono dobbiamo aggiungere il tempo a questa mappa. È il problema dal quale partirà il capitolo 6.

Riferimenti per proseguire

La guida ai sorgenti della raccolta identifica manifesto, componenti e revisioni reali. Il pacchetto didattico contiene sorgenti completi, ingressi e comandi. I frammenti del capitolo provengono da queste due fonti, mantenute distinte.

La documentazione del progetto Swift su [protocolli e controllo degli accessi](#) permette di approfondire requisiti, conformità e visibilità fra moduli. Il nostro esempio introduce soltanto ciò che serve a leggere il confine; le proprietà di concorrenza arriveranno con il problema del prossimo capitolo.

6. Concorrenza, interfaccia e proprietà dello stato

Abbiamo separato i componenti e cambiato il modo di acquisire una fonte senza spostare i controlli del nucleo. Ora lasciamo che il lavoro richieda tempo. Chiedo a Velia di confrontare due contratti nella conversazione dedicata a quel progetto. Mentre aspetto, apro un'altra chat e preparo una nota. La risposta al confronto arriva quando sullo schermo vedo già la nota. Dove deve finire?

La risposta sembra ovvia: nella conversazione del confronto. Eppure basta che il programma usi «la chat selezionata adesso» al momento della consegna per scrivere nel posto sbagliato. Il servizio può avere restituito un risultato corretto, il controllo delle fonti può essere riuscito, e l'applicazione può comunque aver tradito l'incarico.

Questo è un caso preparato per il libro, attraverso il quale leggeremo responsabilità presenti nei sorgenti reali. Non ricostruiamo un incidente documentato di Velia. Partiamo dalle strutture e dai componenti introdotti nei capitoli precedenti; aggiungiamo gli strumenti Swift necessari per ragionare su attese e aggiornamenti. Alla fine dovremo saper distinguere una risposta ancora valida per una chat nascosta da una risposta ormai superata nella stessa chat. Sono due situazioni diverse e richiedono decisioni diverse.

6.1 Guardare altrove non cambia il destinatario

Per capire il primo problema conviene separare due domande: quale conversazione sto mostrando e a quale conversazione appartiene il lavoro? La selezione serve alla prima. Il proprietario dello stato risponde alla seconda. Qui «proprietario» indica il componente incaricato di mantenere e modificare quei dati; non descrive un diritto della persona o la gestione automatica della memoria del linguaggio.

Nella revisione di Velia adottata dal libro, ChatWorkspace mantiene un controller per conversazione. Un controller è un oggetto che coordina lo stato e le operazioni attraverso cui l'interfaccia presenta il lavoro. La proprietà `assistants` associa l'identificatore di una chat al suo `AssistantController`. Questo è il frammento della dichiarazione reale:

```
private var assistants: [UUID: AssistantController] = [:]
```

Le parentesi quadre descrivono un dizionario: a una chiave `UUID` corrisponde un controller. Un `UUID` è un identificatore usato per distinguere entità; nel nostro

ragionamento conta l'uguaglianza dell'identità, non un ordine temporale fra valori. `[:]` crea il dizionario vuoto. La parola `private`, già incontrata nei confini fra componenti, limita l'accesso a questa proprietà secondo le regole di visibilità di Swift.

Quando cambia la selezione, lo spazio delle chat trova il controller da mostrare. Gli aggiornamenti dei controller già creati mantengono invece l'identità della conversazione originaria: i collegamenti che notificano il contenuto acquisiscono quell'identificatore quando vengono costruiti. Non cercano una nuova destinazione nella selezione corrente ogni volta che arriva un pezzo di risposta.

Nel percorso di selezione letto nei sorgenti, passare da una chat all'altra non chiama l'arresto della risposta del controller precedente. Ci sono altre attività interrotte e un salvataggio dello stato della chat lasciata, ma non dobbiamo confonderli con la cancellazione della risposta attiva. Questa scelta permette di tornare al confronto e trovare il lavoro che nel frattempo è proseguito. La figura 6.1 mostra la distinzione da conservare.

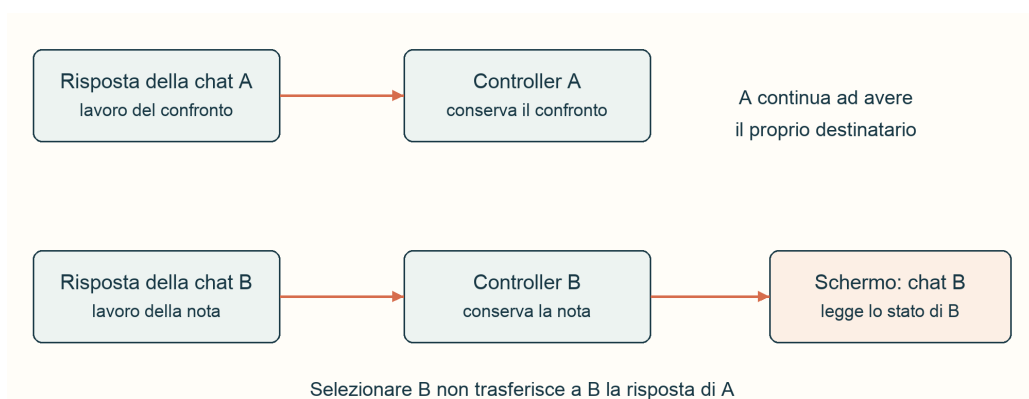


Figura 6.1 - I risultati raggiungono il controller della propria chat. La selezione decide soltanto quale stato leggere sullo schermo.

Se preferissimo un prodotto che interrompe il lavoro quando si cambia chat, dovremmo introdurre quella regola esplicitamente e renderla comprensibile alla persona. Non deriverebbe automaticamente dall'esistenza di una sola finestra visibile. In Velia leggiamo una soluzione concreta; nel laboratorio ne conserviamo il criterio: cambiare selezione non trasferisce un risultato a un altro proprietario.

6.2 Aspettare senza occupare tutto il percorso

La lettura da file del capitolo 5 terminava prima che il chiamante proseguisse. Una risposta remota può richiedere molto più tempo. Se tenessimo occupato il percorso che gestisce l'interfaccia durante tutta l'attesa, la persona potrebbe non riuscire a cambiare chat, premere arresto o anche soltanto vedere un aggiornamento.

In Swift una funzione dichiarata `async` può partecipare a un'esecuzione asincrona. Nel nostro caso può iniziare una lettura, sospendersi mentre il risultato manca e riprendere quando diventa disponibile. Il chiamante scrive `await` nel punto in cui accetta quella possibile sospensione. La sospensione riguarda il lavoro in corso: non impone al programma di fermare ogni altra attività.

La parola «possibile» è necessaria. `await` non è una promessa che il lavoro finirà dopo un certo tempo o che verrà eseguito su un nuovo thread. Un thread è una risorsa di esecuzione del programma gestita dal sistema. Il meccanismo di concorrenza organizza il lavoro e le attese senza chiederci di assegnare manualmente un thread a ogni richiesta.

Concorrenza e parallelismo rispondono a domande diverse. Due lavori sono concorrenti quando i loro percorsi possono avanzare in modo intrecciato: mentre uno attende, l'altro può proseguire. Sono eseguiti in parallelo quando lavorano effettivamente nello stesso intervallo su risorse di esecuzione diverse. Per mantenere utilizzabile una chat durante una lettura ci serve prima di tutto gestire bene l'attesa. Dichiarare una funzione `async` non trasforma automaticamente un lungo calcolo sincrono in lavoro eseguito altrove.

Un `Task` rappresenta un lavoro asincrono gestito dalla concorrenza Swift. Può attendere altre operazioni e può ricevere una richiesta di cancellazione. Non è un agente: non gli attribuiamo capacità di scegliere un piano perché esiste un task. Nella creazione `Task { ... }` le parentesi contengono il lavoro da eseguire. Tenere il valore restituito permette, per esempio, di chiedere successivamente l'arresto con `cancel()` o attendere il risultato con `value`.

Il capitolo introduce questi meccanismi per un problema preciso. Il riferimento primario è il capitolo sulla **concorrenza di *The Swift Programming Language***, mantenuto dal progetto Swift. Spiega sospensioni, task e isolamento del linguaggio; non stabilisce a quale conversazione debba appartenere una risposta della nostra applicazione. Questa decisione resta nel contratto che stiamo costruendo.

6.3 Proteggere l'accesso non basta a scegliere il risultato

Nel capitolo 5 abbiamo incontrato `actor` nel deposito. Un `actor` protegge il proprio stato isolato coordinando gli accessi attraverso le regole di concorrenza. Per lo stato dell'interfaccia Swift offre anche `MainActor`, un ambito di isolamento condiviso adatto agli aggiornamenti che devono svolgersi sul percorso principale dell'applicazione. Scrivere `@MainActor` su una classe assoggetta il suo stato isolato e le operazioni interessate a quell'ambito.

Nel codice reale sia `ChatWorkspace` sia `AssistantController` sono annotati `@MainActor`. Questo ci aiuta a impedire accessi concorrenti non coordinati agli stessi dati. Non dimostra però che ogni risposta arrivata appartenga ancora al tentativo corrente. Per vedere il limite, nel laboratorio ho preparato deliberatamente una classe insufficiente. Il metodo seguente è estratto da quella classe, anch'essa annotata `@MainActor`:

```
func acquisisci(_ id: String, tramite servizio: ServizioControllato) async {
    let risposta = await servizio.leggi(id)
    testo = risposta
}
```

`servizio.leggi(id)` attende un `testo`. Dopo l'attesa il metodo assegna il risultato alla proprietà `testo`. Il metodo non accerta che quel risultato sia ancora quello desiderato. Avviamo una prima acquisizione, poi una seconda. Consegniamo la seconda risposta e, soltanto dopo, la prima: il vecchio `testo` sovrascrive il nuovo.

Non serve che due istruzioni modifichino contemporaneamente la stessa zona di memoria perché questo errore si verifichi. Gli accessi possono essere correttamente isolati e la sequenza essere sbagliata rispetto all'intento. La protezione da una *data race*, cioè da accessi concorrenti incompatibili e non coordinati alla memoria, non sostituisce il controllo della validità logica del risultato.

Che cosa può accadere durante `await`. Un'operazione isolata che si sospende lascia la possibilità ad altro lavoro ammesso sullo stesso `actor` di avanzare. Quando riprende, lo stato può essere cambiato. Questa possibilità è chiamata *rientranza*, o *reentrancy*. La proposta ufficiale [SE-0306, *Actors*](#), di John McCall, Doug Gregor, Konrad Malawski e Chris Lattner, chiarisce perché l'isolamento non rende indivisibile un intero metodo che contiene attese. Per noi la conseguenza è concreta: una condizione controllata prima di attendere può richiedere un nuovo controllo dopo l'attesa.

Anche il tratto sincrono merita attenzione. Se eseguiamo un calcolo molto lungo dentro un metodo di `MainActor`, le altre attività che devono usare quell'ambito dovranno aspettare. Nel laboratorio i tratti isolati confrontano poche identità e aggiornano poche proprietà. Il lavoro pesante richiederà una collocazione progettata e

dati trasferibili appropriati; aggiungere `async` al nome non risolve da solo quel problema.

6.4 Identificare il tentativo che può ancora scrivere

Torniamo alla stessa chat. Il confronto è in corso; lo interrompo e ne avvio un altro. Il vecchio risultato appartiene alla conversazione giusta, ma al tentativo sbagliato. Ci serve dunque un'identità più precisa del solo identificatore della chat.

Nel laboratorio usiamo un valore che porta entrambi i riferimenti:

```
struct RiferimentoTurno: Equatable, Sendable {  
    let chat: String  
    let generazione: UUID  
}
```

`chat` distingue la conversazione; `generazione` distingue un suo tentativo. La struttura usa proprietà `let`, che non possono essere riassegnate dopo la creazione del valore. `Equatable` permette il confronto di uguaglianza secondo i requisiti del protocollo. `Sendable` dichiara al sistema di tipi che il valore può attraversare confini di concorrenza rispettando i controlli previsti; questi campi di valore sono compatibili con quella dichiarazione. Non rende valido il riferimento per sempre e non concede accesso a una chat.

Quando avvia un tentativo, il controller conserva la generazione corrente e restituisce un riferimento con la stessa identità. Il lavoro asincrono acquisisce quel valore e il controller destinatario prima di attendere. Quando arriva la risposta, torna a quel controller con quel riferimento. Questo è il metodo di accettazione del laboratorio, nella classe `Conversazione` annotata `@MainActor`:

```
func accetta(_ risposta: String, per riferimento: RiferimentoTurno) -> Bool {  
    guard riferimento.chat == id,  
          generazione == riferimento.generazione,  
          fase == .attesa else { return false }  
    testo = risposta  
    fase = .completa  
    return true  
}
```

Leggiamolo nell'ordine in cui prende la decisione. `guard` richiede che tutte le condizioni siano vere; altrimenti il metodo restituisce `false` senza sostituire il testo. Prima controlla la chat, poi la generazione, infine che il tentativo stia ancora aspettando una risposta. Soltanto dopo assegna il testo e porta la fase a completa. Il valore `true` informa il chiamante che la consegna è stata accettata.

Fra i confronti e l'assegnazione non c'è `await`. Controllo e modifica si svolgono nello stesso tratto sincrono isolato: un'altra operazione sullo stesso stato isolato non può inserirsi nel mezzo attraverso una sospensione di questo metodo. Se inserissimo una lettura asincrona dopo il `guard`, dovremmo rivalutare le condizioni prima della modifica. Proteggere il controller serve a rendere affidabile quel tratto; scegliere le condizioni serve a renderne corretta la decisione.

Nell'avvio conserviamo il destinatario invece di consultare la selezione alla fine. Nella sostituzione invalidiamo la generazione vecchia prima di creare la nuova. La figura 6.2 segue il caso in cui la risposta precedente arriva per ultima: il suo contenuto non decide se può scrivere, lo decide il confronto del riferimento con lo stato corrente.

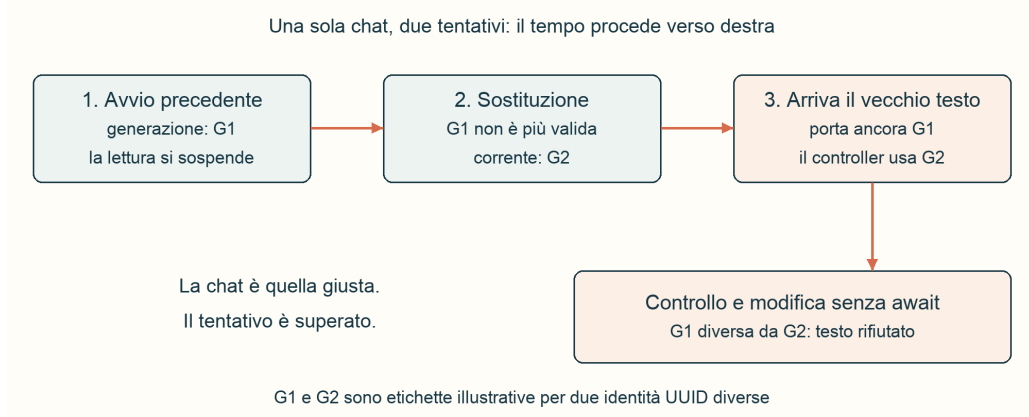


Figura 6.2 - Il nuovo tentativo sostituisce la generazione corrente. La risposta precedente torna alla propria chat, ma il controllo ne rifiuta la generazione. G1 e G2 sono etichette per due identità diverse.

Che cosa identifica una generazione. Qui il valore non è un numero di versione del modello né una revisione del documento. È l'identità del tentativo che il controller accetta. Se cambia il documento mantenendo aperto lo stesso tentativo, questo confronto da solo non rileva il cambiamento: restano necessari i controlli di revisione dei capitoli precedenti. Analogamente, un'identità corretta non sostituisce la verifica degli accessi.

6.5 Come ritroviamo il criterio dentro Velia

Il laboratorio riduce il problema a una risposta finale. Velia riceve anche aggiornamenti intermedi. Nel metodo reale `AssistantController.append`, prima di usare un aggiornamento, compare questa condizione:

```
guard self.generation == generation, phase.isActive, !messages.isEmpty else { return }
```

`self.generation` è la generazione corrente del controller; il parametro `generation` accompagna l'aggiornamento ricevuto. Il controllo richiede anche una fase attiva e la presenza di messaggi. Per un aggiornamento testuale ammesso, il metodo aggiunge poi il frammento al messaggio in costruzione. Il valore acquisito quando è stata avviata la risposta permette di riconoscere gli aggiornamenti che appartengono al lavoro precedente.

Nel percorso di consegna finale troviamo inoltre un controllo della cancellazione e un confronto della generazione prima di applicare il risultato. Anche la gestione dell'errore controlla la generazione: un errore vecchio non deve portare nello stato di errore una risposta nuova. Seguire soltanto il percorso riuscito lascerebbe quindi aperta una seconda porta attraverso cui il vecchio lavoro può modificare la chat.

La stessa esigenza riappare nel deposito, con identità proprie del nucleo. `AgentRuntime` coordina il lavoro degli strumenti; `AgentStore.finishStep` confronta generazione e stato dell'esecuzione e del passo all'interno della transazione locale prima di ammetterne il completamento. Il deposito non delega tutta la decisione a un controllo fatto prima di un'attesa dal chiamante.

C'è una distinzione che conta: in quel metodo può essere registrato il consumo effettivo delle risorse anche dopo una cancellazione, prima del controllo che ammette il risultato come completamento. Rifiutare un risultato superato non significa fingere che il lavoro non sia costato nulla o che nessun dato locale possa più cambiare. Stiamo proteggendo una particolare transizione e il suo risultato, mantenendo separate altre responsabilità. La transazione del deposito, inoltre, non rende reversibile un'operazione già avvenuta su un servizio esterno.

Questi passaggi sono verificabili nella revisione indicata dalla guida ai sorgenti. Li leggiamo per collegare il principio al prodotto reale. Il programma di questo capitolo non compila Jarvis e non certifica che ogni percorso di Velia applichi correttamente lo stesso criterio.

6.6 Fermare è una decisione locale e una richiesta al lavoro

Premere arresto esprime una decisione: questo tentativo non deve più avanzare come tentativo corrente. Riuscire a interrompere immediatamente ogni componente coinvolto è un problema ulteriore. Una risposta potrebbe essere già arrivata al sistema di rete, una libreria potrebbe non controllare la cancellazione o un'operazione esterna potrebbe essere già terminata.

`Task.cancel()` segnala la cancellazione al task. Il lavoro deve cooperare, per esempio controllando la richiesta e terminando o attraversando operazioni che reagiscono alla cancellazione. `Task.checkCancellation()` permette di sollevare un errore quando il task è cancellato; leggere `Task.isCancelled` permette di osservarne il segnale. Nessuna delle due cose annulla una scrittura già avvenuta altrove.

Nel metodo reale `cancelResponse`, Velia cambia la generazione prima di chiamare `task?.cancel()`. L'operatore `?.` esegue la chiamata se il riferimento al task esiste. Il metodo avvia inoltre il percorso che chiede al coordinatore l'arresto del tentativo e la conclusione dell'accesso alle cartelle. Queste operazioni contengono attese: l'invalidazione locale non coincide con la loro conclusione. Il metodo `stop()` non attende il task di cancellazione restituito dal percorso interno.

Nel laboratorio conserviamo la parte essenziale di quest'ordine: `interrompi()` rende invalida la generazione e porta la fase a fermata; poi chiama `cancel()` chi ha conservato il task. Abbiamo preparato un servizio che ignora deliberatamente il segnale e restituisce ugualmente il testo. La risposta arriva, ma accetta la rifiuta. Questa prova serve proprio a non attribuire alla cooperazione del servizio la protezione dello stato locale.

Perché studiare anche un servizio che non collabora. La proposta ufficiale [SE-0304, *Structured Concurrency*](#), di John McCall, Joe Groff, Doug Gregor e Konrad Malawski, descrive la cancellazione cooperativa. Aiuta a capire perché il segnale e la fine effettiva siano distinti. Il nostro servizio controllato forza un caso limite di consegna tardiva; non è il modello da copiare per un trasporto di produzione. Un adattatore reale deve liberare risorse e propagare l'arresto per quanto consentito dal servizio con cui comunica.

Questo ordine è riutilizzabile anche fuori dalle chat: invalidare ciò che non può più scrivere, chiedere l'arresto del lavoro, gestire l'eventuale ritorno tardivo. Se il compito ha già prodotto un effetto esterno, dobbiamo anche accertare e comunicare quell'effetto. Il semplice rifiuto del testo non basta a riportare il mondo alla situazione precedente.

6.7 Provocare l'ordine che mette in difficoltà il programma

Il laboratorio completo richiede Swift 6 e Python 3. È un programma da riga di comando: rappresenta lo stato osservabile da una possibile interfaccia, senza aprire una finestra o chiamare un modello. Le chat A e B sono nomi preparati per l'esperimento, distinti dai documenti A e B usati nei confronti dei capitoli precedenti.

Un servizio controllato riceve le letture e le tiene sospese finché il caso di prova non consegna il testo. Prima di cambiare stato aspettiamo un segnale esplicito che conferma l'avvio della lettura. Non usiamo un'attesa di qualche millisecondo sperando che l'altra attività sia partita: su un computer più lento quella supposizione potrebbe essere falsa.

La continuazione conserva il punto da cui ripartire. Nel servizio il meccanismo `Swift withCheckedContinuation` consente di sospendere l'operazione e ottenere un riferimento attraverso cui riprenderla con un risultato. Il servizio conserva quel riferimento; la consegna lo rimuove e lo riprende una sola volta. Dimenticarlo lascerebbe il lavoro sospeso, riprenderlo due volte violerebbe il contratto. Il driver impone anche una scadenza esterna al processo: un caso bloccato è un errore della prova, non una consegna riuscita.

La tabella indica che cosa osservare nei casi preparati. Le due varianti di ordine per cambio chat e sostituzione portano il totale a otto esecuzioni.

Caso	Che cosa deve rendere osservabile
Risposta corrente	Il proprio controller accetta testo e completamento.
Cambio chat, nei due ordini di arrivo	A conserva il proprio risultato; B conserva il suo e resta la chat visibile.
Tentativo sostituito, nei due ordini	La vecchia generazione viene rifiutata sia prima sia dopo la consegna nuova.
Arresto con servizio non cooperativo	Il task risulta cancellato; il testo tardivo non riapre la chat fermata.
Solo isolamento	La classe insufficiente sovrascrive il testo nuovo con quello vecchio: è l'errore atteso del controesempio.
Due letture indipendenti	Entrambe sono avviate; la consegna invertita conserva le identità dei risultati.

Dalla radice del repository possiamo eseguire tutti i casi:

```
python3 quality-gates/storage/gate.py run -- \
python3 ai-agentica/esempi/velia/capitolo-06/esegui.py \
--verifica
python3 quality-gates/storage/gate.py check
```

Il comando esterno gestisce lo spazio transitorio della prova. Il driver compila con i controlli di Swift 6 e tratta gli avvisi come errori; poi verifica testo, identità, fase e

conclusione delle attese previste. Sorgenti e scenari restano disponibili; eseguibili, cache e risultati vengono rimossi dal ciclo.

Nel caso solo-isolamento il criterio è osservare il difetto intenzionale. Il buon esito di quel caso non significa che abbiamo reso corretta la classe: significa che l'esperimento ne ha mostrato il limite. Nei casi protetti, invece, il criterio è impedire la consegna errata. La distinzione evita di leggere un unico messaggio finale di successo come se certificasse tutte le implementazioni presenti.

Controllare questi ordini non esplora ogni intreccio possibile delle operazioni. Non misura neppure prestazioni o correttezza semantica della risposta. Rende ripetibili alcune sequenze decisive e verifica che la soluzione proposta rispetti in quelle condizioni il proprio contratto.

6.8 Due letture che possono avanzare separatamente

Trasferiamo il ragionamento alla raccolta delle fonti. Se la lettura del secondo documento non richiede il contenuto del primo, possiamo avviarle entrambe prima di attenderne il risultato complessivo. Nel laboratorio il metodo dedicato contiene questo frammento:

```
async let a = servizio.leggi("fonte-A")
async let b = servizio.leggi("fonte-B")
return await (a, b)
```

`async let` avvia un'attività figlia per ottenere ciascun valore. La relazione di parentela significa che la loro durata è governata dall'ambito che le contiene: il percorso non le abbandona come lavori senza più un proprietario. Questo è un caso di concorrenza strutturata. L'ultima riga attende i due risultati e li restituisce in una coppia, mantenendo l'associazione con `a` e `b` anche se `B` finisce prima.

La prova aspetta che entrambe le letture siano avviate e consegna prima `B`, poi `A`. Verifica che il primo elemento della coppia resti il contenuto di `A`. Non usa un cronometro per concludere che il programma sia due volte più veloce, e non deduce che le letture siano state eseguite in parallelo. Quando le operazioni sono realmente indipendenti, questa struttura permette loro di attendere senza imporre che una termini prima dell'avvio dell'altra.

Il valore della coppia è un buon dato da passare al seguito: contiene risultati distinti, senza richiedere che due attività modifichino liberamente un solo testo condiviso. In un trasporto reale porteremmo anche identità e revisioni delle fonti, conservando i controlli del nucleo. Se invece la seconda richiesta dipendesse da una scelta basata sul

primo documento, l'indipendenza sarebbe falsa e dovremmo prima attendere quella scelta.

La creazione con `Task { ... }` usata per i tentativi del laboratorio è diversa da `async let`: produce un task non strutturato, del quale conserviamo esplicitamente il riferimento e attendiamo il risultato nei casi di prova. In un'applicazione quel riferimento richiede una responsabilità chiara per la durata e la cancellazione. Anche le attività figlie, comunque, devono cooperare all'arresto: la struttura non può costringere a terminare un servizio che mantiene una continuazione sospesa senza riprenderla.

6.9 Provare a cambiare la situazione

Immaginiamo di lasciare aperte due chat mentre viene interrotto soltanto il confronto nella prima. Quali dati devono cambiare perché la risposta della seconda resti valida? Prima di toccare il codice, indichiamo il proprietario di ciascun risultato, la generazione da invalidare e il punto in cui verificarla. Se un unico comando cancella tutti i task disponibili, abbiamo introdotto una regola più ampia della richiesta ricevuta.

Un'altra variante conserva il tentativo, ma cambia una fonte prima della consegna. Spieghiamo perché la generazione possa ancora coincidere e il contenuto dover essere rifiutato o qualificato come fotografia di una revisione precedente. Individuiamo quale controllo manca e dove debba stare. Identità del tentativo e revisione del documento risolvono problemi che si incontrano nello stesso percorso senza diventare equivalenti.

Infine sostituiamo il testo finale con una serie di frammenti. Dopo l'arresto ne arriva ancora uno. Il controllo va applicato a ogni aggiornamento che modifica lo stato, oltre che alla consegna finale e all'errore. Questa è la situazione che rende naturale il prossimo passo: collegare Velia al modello e distinguere dati ricevuti, risposta in costruzione e completamento effettivo. I risultati devono continuare ad appartenere al lavoro giusto anche quando arrivano una parola alla volta.

7. Collegare un modello e leggere una risposta progressiva

Nella chat del confronto compare una frase: «La consegna passa da dieci a...». Poi il collegamento si interrompe. Abbiamo ricevuto una risposta? Abbiamo ricevuto dei dati, certamente. Abbiamo anche potuto mostrare alla persona che il lavoro avanzava. Non sappiamo ancora se il servizio abbia concluso la risposta, quali elementi manchino e se il testo visibile coincida con la consegna finale.

Nel capitolo 6 abbiamo stabilito chi può modificare la conversazione e come rifiutare il risultato di un tentativo superato. Ora dobbiamo decidere che cosa stia arrivando a quel proprietario. Una sequenza di byte, un frammento di testo, una richiesta di strumento e una risposta conclusa appartengono allo stesso percorso, ma non sono intercambiabili.

Seguiremo il trasporto reale di Velia nella revisione adottata dal libro. Il laboratorio isolerà la costruzione della richiesta e la lettura di un flusso preparato: potremo tagliarlo nei punti desiderati senza credenziali o chiamate a pagamento. Non sarà una prova della rete o del modello. Ci permetterà di comprendere un confine prima di usarlo nel ciclo del capitolo successivo.

7.1 Quattro scelte dietro «usa questo modello»

Il nome del modello non identifica tutto il collegamento. Il provider è il soggetto che offre il servizio. Il modello identifica la capacità di elaborazione richiesta a quel servizio. L'interfaccia API stabilisce come presentare ingressi e leggere uscite. Il trasporto realizza la comunicazione concreta, con destinazione, autenticazione, tempi ed errori.

Nel percorso che esaminiamo, `OpenAIChatModelTransport` ha come destinazione predefinita l'endpoint `https://api.openai.com/v1/responses`. Un endpoint è la destinazione del servizio che riceve un determinato tipo di richiesta. Velia costruisce il corpo per la Responses API e legge una risposta progressiva. Non possiamo sostituire il nome del provider e presumere che un altro servizio accetti gli stessi campi, restituisca gli stessi eventi o supporti le stesse capacità.

Anche due modelli dello stesso provider possono accettare combinazioni differenti di parametri. Il nome non dimostra che siano disponibili immagini, strumenti, un particolare livello di ragionamento o la stessa dimensione di contesto. La

configurazione reale di Velia conserva modello e modalità di autenticazione; i percorsi verificano alcune incompatibilità e lasciano al servizio la verifica di identificatori sconosciuti. Queste sono scelte del progetto nella sua revisione, non un catalogo aggiornato di ciò che ogni account può usare.

Riferimento essenziale - Il contratto del servizio. La documentazione ufficiale OpenAI su [risposte progressive](#) e [function calling](#), consultata per questa stesura il 28 ottobre 2026, descrive eventi e scambio delle richieste di strumenti. Ci serve per distinguere il contratto pubblico dalle decisioni del client. I sorgenti di Velia restano fissati alla revisione della guida: una nuova pagina della documentazione non aggiorna retroattivamente il codice del prodotto.

Questa distinzione è pratica. Se una chiamata fallisce, dobbiamo capire se il modello manca nell'account, se il corpo contiene un parametro incompatibile, se l'autenticazione non è valida o se il collegamento non raggiunge il servizio. «Il modello non funziona» riunirebbe problemi che richiedono interventi diversi.

7.2 La credenziale viaggia fuori dal contesto

Per ottenere il confronto, il modello ha bisogno dell'incarico e dei dati pertinenti. La credenziale serve invece al servizio per riconoscere l'accesso del client. Il nome token usato per quel segreto non indica qui le unità di testo contate dal modello. Metterla nelle istruzioni del modello confonderebbe le due responsabilità e la esporrebbe senza necessità.

Il metodo reale di trasporto riceve separatamente il corpo body, la credenziale token, la modalità mode, la scadenza deadline e una funzione per gli aggiornamenti. Questo è il requisito dichiarato dal protocollo, riformattato per leggerlo:

```
func complete(  
    body: Data, token: String,  
    mode: AuthenticationMode, deadline: Date,  
    onText: @escaping @Sendable (String) async -> Void  
) async throws -> Data
```

Data contiene byte posseduti dal valore: in ingresso rappresenta il corpo della richiesta; in uscita il documento completo restituito dal percorso. `async throws` permette attesa ed errore, già incontrati nel ponte a Swift. `onText` è una funzione che il trasporto può richiamare con una parte di testo. Il suo risultato Void significa che non restituisce un valore utile al chiamante. La funzione può attendere e viene dichiarata adatta all'uso attraverso i confini di concorrenza con `@Sendable`.

`@escaping` indica che la funzione può essere conservata e usata oltre l'ambito della funzione che la riceve. Non le assegna automaticamente un actor e non decide se il

testo possa ancora entrare nella chat. Il percorso di aggiornamento deve continuare a tornare al controller e ai controlli di generazione studiati nel capitolo 6.

Nel ramo con chiave API, il trasporto inserisce il token nell'intestazione HTTP Authorization, con lo schema Bearer, e il corpo JSON in `httpBody`. Un'intestazione accompagna la comunicazione; non è un messaggio del contesto del modello. Il codice del client comprende `KeychainTokenStore` e un'implementazione di archiviazione del sistema. Il Keychain è il servizio macOS per conservare credenziali; le politiche di accesso e la protezione biometrica hanno percorsi propri. Conservare un segreto nel Keychain non significa che durante l'uso quel valore non debba mai essere presente nella memoria del programma.

Buona pratica - Separare segreti e dati del compito. La guida OpenAI sulle **pratiche di produzione** richiede di proteggere le chiavi ed evitarne l'inserimento nel codice pubblico. Nel nostro percorso distinguiamo anche corpo, intestazioni e diagnostica: un errore del collegamento non giustifica stampare una credenziale. Il laboratorio non ne riceve alcuna. Il fatto che la sua richiesta preparata sia priva di segreti non qualifica tutta la gestione delle credenziali di Velia.

Velia comprende anche una modalità di accesso diversa dalla chiave API. Qui non ne riproduciamo il contratto come se fosse un'autorizzazione universale disponibile a ogni lettore. Seguiamo il ramo necessario a comprendere richiesta e flusso, mantenendo identificabile la modalità descritta.

7.3 Costruire il corpo e distinguere ciò che controlla

Il coordinatore prepara istruzioni, messaggi disponibili e descrizioni degli strumenti ammessi. Nel corpo reale troviamo il modello configurato, `input`, `instructions`, `tools` e le opzioni `stream: true`, `store: false`, `parallel_tool_calls: false`. Nel laboratorio costruiamo lo stesso insieme essenziale, senza riprodurre tutti i campi del prodotto.

`stream` richiede una risposta progressiva. `store: false` riguarda la memorizzazione della risposta per il successivo recupero tramite l'API, descritta nella guida allo **stato della conversazione**; non è una prova dell'assenza di ogni conservazione da parte del servizio, e non governa l'archivio locale di Velia. `parallel_tool_calls: false` esprime una scelta sull'emissione delle richieste di strumenti. Il coordinatore del percorso reale controlla anche che l'output non contenga più di una chiamata. Il valore nel corpo e il controllo del risultato hanno ruoli complementari.

Il trasporto crea una richiesta POST, indica il tipo del corpo come `application/json` e richiede `text/event-stream` in risposta. Usa una sessione effimera, senza cache URL o archivio dei cookie associati alla configurazione letta. Il delegato

`OpenAIRedirectPolicy` rifiuta il reindirizzamento: non segue automaticamente una destinazione diversa. Sono fatti leggibili nel codice; verificare che ogni distribuzione li mantenga richiede prove del client e della rete appropriate.

Una risposta HTTP favorevole indica che la richiesta ha raggiunto un percorso che ha restituito un esito di quel tipo. Non dimostra che il corpo contenga la consegna attesa. Nel metodo reale vengono controllati stato HTTP e tipo del contenuto prima di leggere gli eventi. Per un errore HTTP il programma raccoglie una porzione limitata del corpo d'errore e la traduce nel proprio percorso diagnostico.

Abbiamo così almeno tre controlli distinti: validità della comunicazione, validità della risposta del provider e ammissibilità del risultato per l'applicazione. La figura 7.1 li mette in sequenza. Il confine fra gli ultimi due tornerà nel capitolo 8, quando una risposta tecnicamente conclusa proporrà una funzione da eseguire.

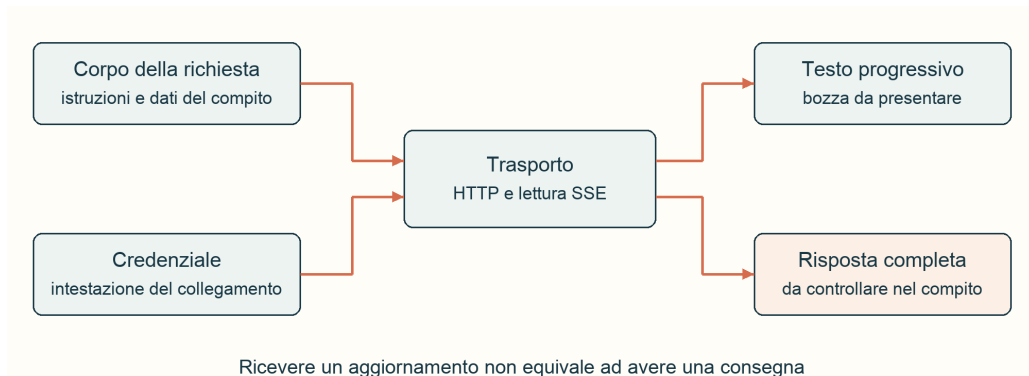


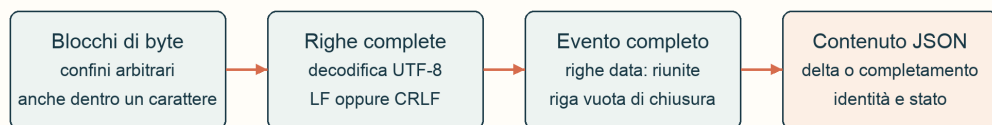
Figura 7.1 - La credenziale serve al collegamento. I dati del compito entrano nel corpo. Ricevere testo e ammettere una risposta completa sono passaggi distinti.

7.4 Dai byte all'evento, dall'evento al testo

Il servizio può inviare una risposta usando Server-Sent Events, abbreviato SSE: un formato testuale che delimita eventi dentro un flusso. Nel sottoinsieme che ci serve, le righe `data:` contengono il contenuto di un evento e una riga vuota ne chiude il blocco. Il contenuto viene poi interpretato come JSON.

Un blocco ricevuto dal collegamento non coincide necessariamente con una riga o con un evento. Potrebbe contenere più eventi, metà di una riga o soltanto una parte dei byte che codificano un carattere. UTF-8, la codifica usata dal flusso, può rappresentare un carattere con più byte. Convertire ogni pacchetto isolato in una stringa completa può quindi danneggiare un testo corretto.

Il laboratorio conserva i byte finché trova il termine di riga; soltanto allora li decodifica. Accumula poi le righe di dati fino alla chiusura dell'evento. Il suo LettoreSSE mantiene questi dati fra una chiamata a ricevi e la successiva. La figura 7.2 mostra perché servano due ricomposizioni, prima delle righe e poi dell'evento.



Il lettore conserva le parti ancora insufficienti fra due consegne

Fine dei byte senza completamento: risposta incompleta

Figura 7.2 - I confini dei blocchi ricevuti non stabiliscono i confini del contenuto. Il lettore conserva le parti ancora insufficienti senza promuoverle a risposta.

Approfondimento - Quale parte di SSE stiamo costruendo. Lo **standard HTML del WHATWG** definisce codifica e interpretazione degli eventi. Il lettore didattico gestisce righe LF o CRLF e dati su più righe; non implementa l'intera interfaccia browser EventSource, riconnessione, identificatori di ripresa o ogni variante del formato. Questi limiti impediscono di presentarlo come parser SSE universale. Le regole del servizio vanno aggiunte sopra il formato generale.

Nel protocollo Responses, il campo JSON type descrive che cosa è successo. response.output_text.delta porta una parte di testo, chiamata delta. response.completed riguarda il completamento della risposta. Altri eventi descrivono errori, incompletezza, rifiuto o avanzamento di elementi diversi. Il nome del campo type non rende sufficiente controllare soltanto la sua esistenza: il contenuto deve corrispondere all'evento atteso.

I delta degli argomenti di una funzione meritano una cautela specifica. Un frammento come {"id": non è una richiesta utilizzabile. Il laboratorio non esegue nulla quando riceve quei delta; rende disponibile la chiamata soltanto leggendo l'output completo. Più avanti potremo studiare ottimizzazioni diverse, ma dovranno conservare una condizione verificabile che impedisca di agire su un argomento ancora incompleto.

Leggiamo adesso i primi due eventi della fixture che rappresenta un'interruzione. Le righe vuote fanno parte del formato e separano i due blocchi:

```
data: {"type":"response.created","response":{"id":"r-1"}}
```

```
data: {"type":"response.output_text.delta","delta":"È una bozza. "}
```

Il primo blocco comunica la creazione della risposta con identità $r-1$. Il secondo porta un delta testuale, «È una bozza.». La riga vuota chiude l'evento di aggiornamento, non l'intera risposta: il fatto che un evento sia completo non significa che sia arrivato l'evento di completamento della risposta. Nella fixture il flusso finisce proprio qui, senza quest'ultimo riscontro.

Possiamo prevedere due esiti distinti prima di eseguire il caso. La funzione di aggiornamento riceverà il testo e potrà renderlo visibile come bozza. Il percorso che aspetta una risposta completa dovrà invece segnalare l'interruzione. Se il programma restituisse la bozza come risultato riuscito, avremmo localizzato l'errore nella promozione del contenuto, senza dover discutere quanto la frase fosse plausibile.

La variante con una funzione cambia il contenuto degli eventi, ma conserva il criterio. Un evento che conclude un pezzo degli argomenti non conclude necessariamente la risposta che li contiene. Nel nostro incremento aspettiamo l'output finale prima di costruire una `Chiamata`. Il ciclo del capitolo 8 riceverà quindi una proposta completa, sulla quale dovrà ancora controllare capacità, identità e accessi.

7.5 Completamento del provider e bozza sullo schermo

Nel parser reale `ChatAgentStreamParser`, l'evento di completamento richiede lo stato `completed`, legge gli elementi finali, confronta l'identità della risposta e riconcilia gli elementi ricevuti durante il flusso con quelli finali. Gli elementi in conflitto o rimasti incompleti incontrano rifiuti specifici. Il coordinatore riceve poi quel documento completo.

Il laboratorio adotta un contratto più piccolo. Richiede prima l'identità della risposta, poi un completamento con la stessa identità, stato concluso e output del sottoinsieme ammesso. Accetta messaggi testuali, chiamate complete e dati reasoning conservati senza interpretarli. Non riproduce il registro degli indici degli elementi del parser di Velia. Il controllo decisivo alla fine dei byte è molto breve:

```
func termina() throws {  
    guard concluso else { throw ErroreFlusso.incompleta }  
}
```

La fine del flusso non imposta `concluso`. È l'elaborazione di un evento finale valido a impostarlo. Se il collegamento finisce dopo una frase plausibile ma prima di quell'evento, `termina` segnala incompletezza. Non stima quanto sia probabile che mancassero soltanto poche parole.

Nella fixture positiva i delta compongono «È una bozza.» mentre il testo conclusivo è «È il testo finale.». La differenza è voluta: il consumatore deve usare la consegna

completa come valore autorevole del percorso, senza presumere che basti concatenare ciò che è già comparso. Nel prodotto reale abbiamo letto un aggiornamento `resetText` prima della presentazione del testo finale elaborato dal coordinatore. Anche eventuali correzioni o qualificazioni della consegna devono arrivare alla chat giusta.

Concetto chiave - Un dato opaco conserva una funzione senza diventare una spiegazione. Alcune risposte comprendono elementi `reasoning`, che possono contenere dati cifrati da restituire nel seguito della conversazione. Il laboratorio conserva l'oggetto serializzato senza cercare di decifrarlo. La sua presenza non è una trascrizione verificata delle cause interne della scelta e non deve diventare una prova scientifica del ragionamento del modello. Conservare il dato utile al protocollo e interpretare il comportamento sono lavori diversi.

Anche un rifiuto va distinto dal normale risultato del compito. Il laboratorio lo segnala come esito separato e non lo ammette come nota del confronto. Il codice reale ha una politica di presentazione più ampia: il parser può inoltrare delta di rifiuto e il metodo `outputText` comprende anche quel contenuto. Non attribuiamo al prodotto la politica restrittiva scelta per il nostro esperimento. Nei due casi rimane necessario evitare che un messaggio di rifiuto sia considerato prova di avere svolto il confronto.

7.6 Scadenza, cancellazione e riconnessione non sono sinonimi

La scadenza stabilisce fino a quando ammettere il lavoro; la cancellazione esprime una richiesta di arresto; la chiusura della connessione è un fatto del trasporto. Il servizio potrebbe aver già elaborato parte della richiesta prima di uno qualunque di questi eventi. Per questo un nuovo invio non è automaticamente la continuazione del precedente.

Il trasporto reale configura tempi della sessione, controlla cancellazione e scadenza durante la lettura dei byte e invalida la sessione nel gestore di cancellazione. Il laboratorio controlla queste condizioni fra i blocchi preparati. Non modella una connessione che rimane silenziosa per minuti: quel caso richiede una prova del livello di rete e dei suoi timeout. Un controllo scritto dentro il ciclo dei byte non può essere eseguito finché non arriva un byte e il ciclo non avanza.

Il lettore didattico impone inoltre un limite totale di 524.288 byte per flusso, scelto per contenere questo esperimento. Non è la dimensione del contesto del modello né un limite documentato del servizio. Un contatore nel sorgente non costituirebbe da solo un limite: occorre un confronto che rifiuti il superamento. La prova dedicata riduce il massimo a 32 byte e controlla quel rifiuto.

Se il flusso si interrompe, possiamo conservare sullo schermo una bozza identificata come interrotta, oppure rimuoverla secondo il contratto di presentazione. Non possiamo promuoverla silenziosamente a risposta conclusa. Se vogliamo riprovare, dobbiamo stabilire quali ingressi rimandare, quale identità attribuire al nuovo tentativo e come trattare eventuali operazioni già avvenute. Nel capitolo successivo il problema diventerà più evidente perché la risposta potrà avere richiesto uno strumento.

7.7 Ripetere i casi senza chiamare un servizio

Il laboratorio del capitolo 7 contiene il lettore, il costruttore della richiesta e un trasporto preparato. Le fixture SSE sono sorgenti curati, scritti per il libro: non registrazioni di un account. Il trasporto consegna i loro byte in blocchi scelti e richiama la funzione che raccoglie la bozza. Richiede Swift 6 e Python 3.

Dalla radice del repository:

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-07/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

I tredici casi comprendono blocchi da un byte e da diciassette byte, righe CRLF, dati su più righe, chiamata completa dopo argomenti frammentati, troncamento, errore del provider, risposta incompleta, cambio d'identità, JSON errato, rifiuto, scadenza superata e limite di byte. La consegna di un byte per volta attraversa anche l'interno del carattere accentato: il testo finale deve restare uguale a quello consegnato con blocchi più grandi.

Il controllo distingue bozza e finale, conserva i dati opachi e verifica che i delta degli argomenti non siano diventati chiamate eseguibili. Il programma non apre una connessione HTTP. Questi casi qualificano il lettore e il contratto didattico nei casi preparati, senza dimostrare autenticazione, disponibilità dell'account, comportamento di proxy o qualità di un modello reale.

Una prova successiva del collegamento dovrà controllare separatamente destinazione, intestazioni senza esposizione dei segreti, tipo del contenuto, errori HTTP e arresto durante un'attesa di rete. Una valutazione del modello dovrà invece confrontare richieste, fonti, risposte e criteri su casi che ne mostrino la variabilità. Separare le domande permette di capire che cosa correggere quando una prova fallisce.

7.8 Una variante da portare al ciclo

Immaginiamo che la connessione termini dopo avere mostrato il confronto ma prima del completamento. La persona chiede di salvare quel testo come documento. Descriviamo quale stato permetterebbe di proporre il salvataggio di una bozza interrotta senza dichiararla un confronto verificato. Il consenso al salvataggio riguarda il documento che abbiamo davvero, non dimostra che il servizio avesse finito.

Cambiamo poi il flusso: arriva una richiesta di lettura di B, seguita dal completamento del provider. Indichiamo quali controlli rimangano prima dell'operazione e quali dati debbano tornare alla richiesta successiva. Avere una chiamata completa risolve il problema di leggere il protocollo; non stabilisce ancora accessi, revisione della fonte o risultato del compito.

Questo è il punto di arrivo del capitolo. Abbiamo un modo per distinguere aggiornamenti e risposta ammessa dal trasporto. Ora possiamo collegare una proposta completa a uno strumento, riportarne l'esito al modello e decidere quando fermare l'intero lavoro.

8. Il primo ciclo eseguibile di Velia

Il trasporto ci restituisce una risposta conclusa. Dentro troviamo una richiesta: leggere la fonte A nella revisione r1. Non abbiamo ancora il confronto. Abbiamo una proposta di operazione dalla quale può nascere un nuovo dato. Il passaggio che rende agentic il percorso sarà restituire quell'esito al modello e permettergli di scegliere il seguito.

Se leggessimo sempre A, poi B, poi chiamassimo il modello per scrivere la nota, costruiremmo il workflow già discusso nel capitolo 2. Qui il ciclo non contiene quell'elenco obbligatorio di letture. Riceve una proposta strutturata, la controlla, esegue una capacità ammessa e prepara una nuova richiesta con ciò che è accaduto. La differenza riguarda chi sceglie il prossimo passo; il programma continua a possedere accessi, stato e condizioni di arresto.

Velia realizza questa alternanza nel coordinatore della chat e usa il nucleo per eseguire e registrare i passi. Ne seguiremo i confini. Poi costruiremo un incremento didattico completo, eseguibile con gli strumenti Swift già usati. Il modello sarà sostituito da sequenze preparate: potremo verificare il programma e provocarne gli errori senza confondere questa prova con una dimostrazione della capacità del modello di scegliere bene.

8.1 Dare una forma al lavoro prima del ciclo

Conserviamo l'incarico: confrontare i tempi di consegna delle fonti A e B nella revisione r1 e sostenere la nota con passaggi ritrovabili. L'assistenza e gli altri contenuti del confronto iniziale restano fuori dal perimetro di questo incremento. Una fonte non accessibile deve diventare un limite dichiarato, senza essere inventata. Le risorse disponibili al laboratorio sono due piccoli testi in memoria, identici per funzione ai dati curati dei capitoli precedenti.

Prima di scegliere classi, distinguiamo tre stati delle informazioni. Una fonte può non essere stata ancora richiesta; può essere stata acquisita; oppure la sua lettura può avere prodotto un accesso negato. L'ultima condizione non equivale alla prima. Se non abbiamo mai tentato di leggere B, non possediamo ancora un riscontro sul quale fondare l'affermazione «l'accesso a B è negato».

Nel laboratorio un'osservazione conserva identificatore, revisione, stato e testo eventuale. La parola osservazione indica qui l'esito ricevuto da una capacità del programma. Non gli attribuisce un'autorità maggiore di quella della sua origine: il catalogo preparato può attestare la propria lettura, non la verità del contratto commerciale rappresentato dal testo.

Una risposta finale deve inoltre indicare che cosa manca. «Parziale» senza una lacuna riconoscibile sarebbe un'etichetta poco utile. Dobbiamo poter collegare la mancanza a una fonte richiesta e a un esito del lavoro. Questa scelta costruisce il criterio di completamento prima del messaggio che annuncia di avere finito.

8.2 Dove vive l'alternanza nel progetto reale

In `ChatAgentCoordinator.respondWithMemory` troviamo un ciclo che prepara capacità e istruzioni, invia una richiesta e legge gli elementi dell'output completo. La chiamata al modello attraversa essa stessa il percorso di esecuzione come capacità `chat_model`, con deposito e conteggio dei consumi. Non è un'attività estranea ai limiti del compito soltanto perché non modifica un file.

Il passaggio reale che distingue le richieste di funzione è questo:

```
let calls = outputs.filter { $0["type"] as? String == "function_call" }  
guard calls.count <= 1 else { throw ChatAgentError.invalidResponse }
```

`filter` seleziona elementi strutturati dell'output: il programma riconosce il tipo previsto dal protocollo, non cerca la parola «leggi» nella frase dell'utente. Il secondo controllo rifiuta più di una richiesta di funzione in quell'output. È una politica del coordinatore letto, coerente con l'opzione della richiesta; non è una proprietà di tutti i sistemi agentici o di tutti i modelli.

Il coordinatore conserva gli elementi completi ricevuti, compresi quelli opachi, per costruire il seguito. Se c'è una funzione, controlla nome, identificatore della chiamata e argomenti, costruisce la capacità eseguibile e usa `AgentRuntime`. Il risultato diventa un elemento `function_call_output`, associato al `call_id` della proposta. Alla nuova iterazione entra quindi un fatto in più: ciò che lo strumento ha restituito.

Se non c'è una funzione, il percorso di consegna applica altri criteri e sceglie la transizione dell'esecuzione. Nel codice reale sono distinti lo stato `completed` del provider e gli stati applicativi `completed` o `partial`. Il secondo dipende anche da passi, operazioni ed evidenze. Il completamento del provider chiude una risposta; il completamento dell'incarico richiede condizioni più ampie.

Concetto chiave - Il ciclo non appartiene a una sola classe per definizione. Il target `VeliaCore` contiene il runtime degli strumenti e il deposito; il coordinatore della chat appartiene al desktop nella revisione studiata. Chiamare entrambi «nucleo agentic» nel discorso generale non li trasforma in un unico modulo. La mappa del capitolo 5 serve anche a ritrovare questa distribuzione concreta.

8.3 Proposta, capacità disponibile e operazione ammessa

Il modello può proporre `leggi_fonte` perché il programma gli ha descritto quella capacità. La descrizione comprende nome, scopo e schema degli argomenti. Nel laboratorio sono ammessi soltanto un identificatore A o B e la revisione `r1`. Lo schema richiede entrambi i campi ed esclude proprietà aggiuntive. Il catalogo eseguibile conserva comunque i propri controlli: ricevere una proposta conforme alle aspettative del modello non equivale ad avere un permesso.

Un tool è dunque più di un nome in una richiesta. Occorre una funzione concreta, argomenti leggibili, una risorsa ammessa, un esito e un comportamento in caso di errore. Un'eventuale skill potrebbe insegnare come confrontare le fonti e trattare le eccezioni, ma non potrebbe concedere accesso a B. Qui non carichiamo ancora un pacchetto di skill: conserviamo la distinzione che servirà nei capitoli dedicati.

Nel nostro `CatalogoFonti`, la lettura è ammessa soltanto dopo il controllo del nome e dei campi. Questo frammento mostra il confine prima dell'accesso al contenuto:

```
guard chiamata.nome == "leggi_fonte" else { throw ErroreCiclo.strumento }
guard let dati = chiamata.argomenti.data(using: .utf8), dati.count <= 4096,
    let campi = try JSONSerialization.jsonObject(with: dati) as? [String: String],
    Set(campi.keys) == ["id", "revisione"],
    let id = campi["id"], let testo = fonti[id],
    campi["revisione"] == "r1" else { throw ErroreCiclo.argomenti }
```

Il primo guard impedisce che un nome diverso scelga una funzione arbitraria. Il secondo impone dimensione, struttura esatta degli argomenti, presenza nel catalogo e revisione richiesta. Un campo come percorso, anche se contiene una destinazione plausibile, viene rifiutato: il catalogo non è un lettore di qualunque file del computer. Set costruisce un insieme, usato qui per confrontare i nomi dei campi senza dipendere dal loro ordine nel JSON.

Solo dopo quei controlli il catalogo registra un tentativo di lettura e applica la regola di accesso preparata. Se B è negata, restituisce un'osservazione senza testo e con stato `accesso_negato`. Il chiamante non può aggirare quel risultato cambiando il testo delle

istruzioni. Questa è una protezione del catalogo del laboratorio; una risorsa reale richiede controlli nel componente che possiede effettivamente i dati.

Nel prodotto, `AgentRuntime` risolve il nome fra capacità registrate e controlla lo schema d'ingresso prima di chiedere al deposito l'avvio del passo. `ChatNativeTool` partecipa ai controlli sugli accessi anche nel proprio percorso. Leggere questi punti evita di confondere la descrizione inviata al provider con l'autorità di eseguire l'operazione.

8.4 Restituire al modello ciò che è successo

La prima risposta preparata propone di leggere A. Il catalogo restituisce il suo testo con identità e revisione. Prima della seconda chiamata il programma conserva sia la proposta completa del modello sia l'osservazione. L'ordine costruisce una relazione: questa è la richiesta, questo è il suo risultato.

Il campo `call_id` identifica la particolare chiamata, non la conversazione o il documento. Se due letture chiedono A, possono avere identificatori di chiamata diversi. Se viene ripetuto lo stesso identificatore in questo ciclo, lo trattiamo come un errore prima di eseguire di nuovo. Questa scelta evita una ripetizione involontaria locale; non realizza da sola l'idempotenza durevole di un'azione remota.

Nel laboratorio la registrazione del risultato nel contesto successivo usa questa forma:

```
elementi.append([
    "type": "function_call_output",
    "call_id": chiamata.id, "output": risultato
])
```

`risultato` è la serializzazione dell'osservazione. Se B è negata, contiene la negazione, non un documento vuoto e non un testo immaginato. La seconda o terza richiesta può quindi ricevere un'informazione che cambia il seguito. La figura 8.1 mette in evidenza il ritorno che distingue questa costruzione da una sola chiamata al modello.

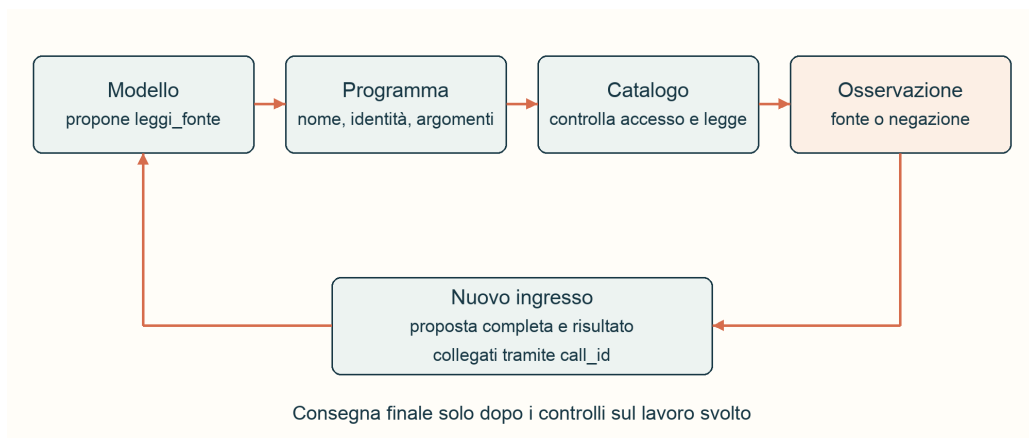


Figura 8.1 - L'esito dello strumento entra nella nuova richiesta con il riferimento alla chiamata. Il modello può proporre il seguito; il programma conserva l'autorità di ammetterlo.

Riferimento essenziale - Il seguito di una chiamata di funzione. La guida OpenAI sul **function calling** distingue la proposta del modello dall'esecuzione nell'applicazione e descrive il ritorno tramite `call_id`. Qui applichiamo quel contratto a letture locali controllate. I risultati del catalogo non dimostrano che un modello reale sceglierebbe le stesse letture o reagirebbe correttamente a una negazione: questa è una domanda della valutazione del modello.

Il codice confronta nomi e campi perché sono parti di un protocollo. Non classifica intenzioni linguistiche. Il testo «Se una lettura è negata, dichiara il limite» appartiene invece alle istruzioni per chi propone il seguito. Nel laboratorio quella decisione è preparata in una fixture; quando useremo un modello reale dovremo verificarla anche su richieste formulate in modi diversi.

8.5 Governare il lavoro durante le attese

Il ciclo attende il trasporto e lo strumento. In quei punti la persona potrebbe avere fermato il tentativo. Riutilizziamo quindi il criterio del capitolo 6: un actor possiede fase, generazione e osservazioni, mentre il lavoro conserva il riferimento ottenuto all'avvio.

Nel laboratorio il metodo `registra` è sincrono rispetto all'isolamento dell'actor:

```

func registra(_ dato: Osservazione, per riferimento: UUID) throws {
    try controlla(riferimento)
    osservazioni[dato.fonte] = dato
}
  
```

controlla richiede che generazione e fase siano ancora quelle ammesse. Fra il controllo e la registrazione non c'è un'attesa. Il ciclo controlla anche dopo il ritorno del modello, prima di accettarne la proposta. La cancellazione del task e l'invalidazione dello stato rimangono strumenti diversi; una nuova attività non deve poter usare il riferimento di quella fermata.

Questo actor mantiene soltanto dati in memoria. Chiudere il processo li perde. Il deposito reale di Velia ha responsabilità di persistenza che questo incremento non ricostruisce: non presentiamo un metodo protetto da isolamento come una soluzione di recupero dopo un arresto del computer. Il lavoro di oggi è rendere comprensibile l'alternanza, mantenendo i confini che conosciamo già.

Anche una proposta sempre formalmente valida può non portare alla conclusione. Per esempio, il modello potrebbe chiedere A senza mai acquisire B. Nel laboratorio consentiamo al massimo tre chiamate al modello: due opportunità di lettura e una di consegna nel percorso preparato. Il ciclo non impone che vengano usate così; al terzo turno ancora privo di consegna si ferma con un limite.

Tre è un parametro dell'esperimento, non una raccomandazione generale per Velia. Non descriviamo il `while true` del prodotto come un ciclo senza limiti soltanto perché non contiene questo stesso contatore. Nel percorso reale il runtime e il deposito governano anche scadenze, risorse e consumi. Le condizioni effettive vanno ricostruite attraverso i componenti che le applicano.

8.6 Quando una risposta può diventare una consegna

Nella terza risposta preparata non c'è una funzione. Il messaggio contiene una nota JSON con stato dichiarato, testo, citazioni e fonti mancanti. Il programma non prende la parola «completo» come prova. Cerca invece le osservazioni che dovrebbero sostenerla.

Il contratto didattico richiede un esito per entrambe le fonti. Da questi esiti ricava quali siano state negate. Una nota completa è ammissibile soltanto se nessuna fonte richiesta manca; una nota parziale deve dichiarare esattamente quelle mancanti. Per ciascuna citazione il controllo richiede una fonte effettivamente acquisita, la stessa revisione e un estratto presente nel testo letto. Ogni fonte acquisita deve avere un riferimento nella consegna di questo piccolo confronto.

La figura 8.2 mostra i due percorsi. Dopo una negazione di B, una nota parziale con la lacuna corrispondente può essere una consegna utile. Una nota che dichiara di avere confrontato entrambi i documenti incontra invece un rifiuto. Lo stato applicativo deriva dal lavoro e dal contratto, non dall'ottimismo della frase.

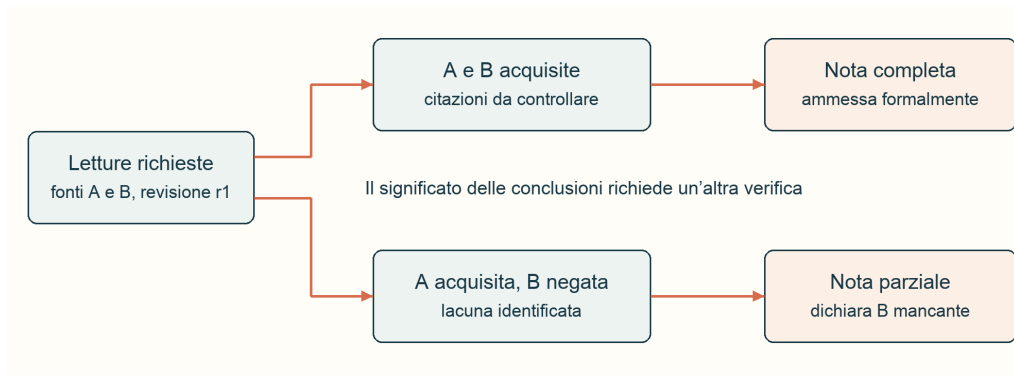


Figura 8.2 - Completare la risposta del provider non significa completare il confronto. Le osservazioni disponibili determinano quali consegne possono essere ammesse.

Rimane il limite del capitolo 4. Un estratto corretto può accompagnare una conclusione sbagliata. Il caso preparato significato-non-valutato cita le frasi dei dieci e quindici giorni, ma dichiara che il termine si riduce a cinque. Il controllo dei riferimenti può ammetterlo. Non ha una regola capace di capire il significato della frase né di verificare che tutte le condizioni commerciali siano state conservate.

Chiamare la fase finale «validazione» non colma quel vuoto. Per valutare il significato servono un controllo adeguato alla domanda e casi capaci di smentirlo. Potrebbe intervenire un revisore umano o un altro modello, ma il nome «revisore» non ne dimostrerebbe l'affidabilità. Dovremmo verificare anche quel giudizio sulle conclusioni corrette, errate, ambigue e incomplete.

Approfondimento - Quattro significati di «finito». Il collegamento può avere smesso di inviare byte; il provider può avere completato la risposta; lo strumento può avere terminato un'operazione; l'incarico può avere soddisfatto i propri criteri. Questi eventi possono avvenire in momenti diversi. Nel laboratorio distinguiamo soprattutto risposta conclusa e nota ammissibile secondo controlli formali. Non deduciamo da questi ultimi la comprensione del contenuto o la soddisfazione completa della persona.

8.7 Leggere il ciclo intero nei sorgenti allegati

Il programma del capitolo 8 riusa `Flusso.swift` del capitolo 7 senza copiarlo. `PrimoCiclo` conserva gli elementi della conversazione e l'insieme degli identificatori di chiamata già incontrati. A ogni turno costruisce l'ingresso, attende una risposta completa, verifica che il lavoro sia ancora attivo e distingue consegna e richiesta di strumento.

La consegna passa al controllo dello stato. La richiesta di strumento attraversa il catalogo e torna come osservazione. Un errore strutturale interrompe il laboratorio;

una lettura riconosciuta ma negata produce invece un esito da usare nel seguito. È una politica esplicita dell'esperimento. Il codice reale di Velia può trasformare diversi errori degli strumenti in osservazioni unavailable, mantenendo un percorso distinto per arresto e altri fallimenti.

La verifica del laboratorio controlla anche l'ingresso successivo: dopo la lettura A devono essere presenti il suo `function_call_output`, il riferimento c-A e gli elementi opachi della risposta precedente. Dopo la negazione di B, il turno successivo deve ricevere `accesso_negato` con il riferimento c-B. Verificare soltanto il testo stampato alla fine non dimostrerebbe quel ritorno.

Dalla radice del repository:

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agantica/esempi/velia/capitolo-08/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

Le nove sequenze hanno criteri diversi, riuniti nella tabella. Il conteggio delle letture è osservato sul catalogo, così un rifiuto tardivo non può nascondere un'operazione già eseguita fuori contratto.

Situazione preparata	Criterio da verificare
Confronto completo	Due letture, osservazioni nel seguito, nota ammessa come completa.
Lettura di B negata	Il turno seguente riceve la negazione; la nota dichiara B mancante e rimane parziale.
Completezza inventata	B è negata ma il testo dichiara completo: consegna rifiutata.
Consegna anticipata	Il modello propone la nota senza acquisizioni: nessuna lettura, consegna rifiutata.
Strumento ignoto o argomenti estranei	Nessuna lettura prima del rifiuto; sono due sequenze distinte.
Identificatore duplicato	La seconda proposta non produce una seconda lettura.
Limite dei turni	Tre proposte ammesse non producono una quarta chiamata al modello.
Significato non valutato	Riferimenti validi e frase sbagliata mostrano il limite del controllo formale.

Le fixture preparano le scelte; il programma realizza realmente attesa, riconoscimento della chiamata, lettura locale, registrazione e nuovo ingresso. La conclusione sostenuta è quindi circoscritta: il ciclo può essere esercitato e controllato con queste decisioni. La capacità di un modello reale di prendere quelle decisioni rimane da valutare, così come il prodotto completo richiede le proprie verifiche.

8.8 Trasferire il ciclo senza trasferire le supposizioni

Aggiungiamo una fonte C che contiene un'eccezione alla consegna. Prima di cambiare lo schema del tool, decidiamo se C sia obbligatoria dall'inizio o se debba essere individuata dopo una lettura. Nel primo caso il contratto di consegna deve comprenderla. Nel secondo serve una regola per acquisire un riferimento emerso dal lavoro mantenendo accessi e identità verificabili. Aumentare il numero massimo dei turni non risolve da solo questa scelta.

Consideriamo poi un'azione diversa: salvare la nota. Il risultato della funzione deve permettere di accertare quale file e quale revisione siano stati scritti. Se il collegamento si interrompe dopo il salvataggio, ricreare il ciclo in memoria non rende innocua la ripetizione. La protezione locale dagli identificatori duplicati non sopravvive al riavvio e non sostituisce il riscontro sulla risorsa esterna. Il cambiamento di azione richiede nuove prove, anche se lo schema generale del ciclo resta utile.

Infine fermiamo il lavoro durante una lettura. Indichiamo chi invalida il tentativo, chi chiede l'arresto al task e quale componente deve rifiutare il dato tardivo. Distinguiamo ciò che possiamo impedire nella consegna da ciò che potremmo dover accertare fuori dal processo. La soluzione ragionata degli apparati propone un criterio, senza dichiarare già costruito il salvataggio remoto.

Abbiamo ora collegato i primi mattoni: un incarico con criteri, componenti con responsabilità distinte, un'attesa che conserva il proprietario, una risposta conclusa e un ciclo che usa gli esiti. Nel capitolo successivo entreremo nella capacità eseguibile stessa: come una funzione diventa un tool con argomenti, risultati ed errori abbastanza precisi da poter essere usato e controllato.

Apparati - Soluzioni, glossario e percorsi di approfondimento

Le soluzioni seguenti sono ragionate: mostrano criteri e limiti, senza imporre un'unica architettura quando il problema ammette alternative. Le attività sono riprese dai capitoli 1-8. Per usarle come esercizi conviene formulare prima la propria risposta, poi confrontare che cosa ciascuna soluzione rende osservabile.

Salvare il confronto come PDF

Nel capitolo 1 il risultato cambia da testo nella conversazione a documento salvato. Il contenuto pronto è una prima condizione, ma dobbiamo anche sapere quale revisione delle fonti descriva. La conversione produce una particolare revisione del PDF; il salvataggio deve riguardare quei byte e una destinazione ammessa. Il messaggio del modello che annuncia un file non sostituisce il riscontro dell'operazione.

Se la cartella non è scrivibile, la consegna può conservare il testo e indicare il problema, oppure chiedere una destinazione consentita. Non deve dichiarare il file salvato. Se una fonte cambia dopo la conversione, occorre decidere se il PDF resti una fotografia identificata della revisione precedente o se vada rigenerato per soddisfare l'incarico corrente. La decisione dipende dal criterio ricevuto e deve essere esplicita.

Una prova utile osserva contenuto, revisione, destinazione ed effetto. Una variante interrompe il programma dopo la scrittura e prima della sua conferma: ripetere il salvataggio senza sapere che cosa sia avvenuto non è una soluzione generale. Può servire verificare il file atteso o ricostruire l'operazione con un identificatore.

Scegliere fra chiamata, workflow e agente

Per la chiamata isolata possiamo usare due estratti già preparati e una richiesta di spiegazione della differenza. L'acquisizione è stata svolta fuori dalla chiamata e il controllo valuta la fedeltà del confronto. Se manca una fonte, la soluzione non deve inventarla.

Per il workflow possiamo usare un controllo periodico di file con struttura stabile: acquisire i due file, estrarre cinque campi noti, verificare la disponibilità e chiedere al modello una nota. La procedura può avere rami per errore e dati mancanti. La sua adeguatezza deriva dalla stabilità del percorso e dai criteri definiti, non dall'assenza di un modello.

Per il ciclo agentic possiamo usare una ricerca documentale nella quale le discrepanze richiedano fonti diverse. Il modello propone il seguito dopo le letture, mentre il programma mantiene accessi, identità e condizioni di arresto. Un limite possibile è la mancanza di una fonte obbligatoria; il risultato previsto può essere una domanda o una consegna parziale. La flessibilità va confrontata con una procedura più semplice sugli stessi casi.

Più candidati alla ripresa

L'estensione del capitolo 3 richiede prima un contratto del perimetro. Il programma può rendere disponibili più candidati ammessi della conversazione o del progetto autorizzato, conservando identità, stato e riferimenti alla richiesta originaria. Il numero e il tipo dei candidati devono essere motivati dalla capacità e dagli accessi, non scelti cercando parole nel testo dell'utente.

Il modello valuta la pertinenza dei metadati rispetto alla richiesta corrente. Se due lavori restano compatibili con ciò che la persona ha detto, una domanda è un risultato valido. La richiesta di recupero deve indicare un'identità esatta e il programma deve ricontrollarne ammissibilità e accessi al momento della lettura. Il risultato resta un insieme di riferimenti del passato: non importa permessi e non riesegue azioni.

Una prova del programma può usare una scelta preparata con un identificatore di un'altra conversazione e richiederne il rifiuto. Una valutazione del modello può usare richieste parafrasate o ambigue e giudicare scelta o chiarimento. Le due famiglie di prove rispondono a domande diverse.

Una terza fonte e una verifica insufficiente

Per l'allegato C del capitolo 4 aggiungiamo un'identità attesa e una fonte disponibile con revisione. Un'affermazione sull'assistenza può indicare una citazione di C. Il programma può controllare che la fonte esista, che la revisione coincida e che l'estratto sia presente. Se C manca, la fonte attesa resta assente.

Il controllo può comunque riuscire se il testo omette un'eccezione decisiva presente in un altro paragrafo. Un controllo aggiuntivo deve ricevere domanda, affermazione, passaggio citato e contesto pertinente, e verificare se condizioni ed eccezioni cambino la conclusione. Un giudizio del modello rimane da valutare con casi curati e confronti; la parola «revisore» non ne rende automaticamente affidabile l'esito.

Un'altra estensione riguarda l'identità della versione. Le etichette delle fixture non provano l'integrità dei byte. Per collegare revisione e contenuto serve definire chi attribuisce l'identità e come un'impronta venga associata alla fonte acquisita.

L'impronta può rilevare una differenza di contenuto, ma non stabilisce da sola l'autore o la verità del documento.

Cambiare il trasporto e conservare le responsabilità

Nel capitolo 5 il trasporto restituisce una fonte, mentre il nucleo verifica che identità e revisione siano quelle richieste. Questa distribuzione va conservata passando a una lettura remota. Il risultato deve permettere di identificare la fonte; un errore deve distinguere almeno assenza, accesso negato e impossibilità di raggiungere il servizio. Il trasporto può tradurre le risposte tecniche in questi esiti, senza trasformare una connessione fallita in un documento vuoto.

Il protocollo sincrono del laboratorio non esprime il lavoro nel tempo. Prima della rete dobbiamo decidere come attendere senza bloccare l'interfaccia, come trasmettere una scadenza e come chiedere l'annullamento. Il coordinatore mantiene l'identità dell'incarico che usa la lettura; il trasporto deve reagire alla cancellazione per quanto il collegamento consente. Una risposta tardiva non deve aggiornare automaticamente la nuova conversazione o un incarico diverso. I controlli di identità e revisione rimangono necessari anche dopo una connessione tecnicamente riuscita.

Una verifica utile provoca il cambio di conversazione durante l'attesa e osserva dove arriva il risultato. Un'altra restituisce la revisione r_2 pur avendo richiesto r_1 : il nuovo trasporto deve continuare a incontrare il rifiuto del nucleo. Il laboratorio locale non esegue queste letture di rete o prove temporali; sono criteri per progettare l'estensione, che richiede le nozioni del capitolo successivo.

La variante di presentazione ha un altro proprietario. Per mostrare ciascuna fonte su una riga distinta possiamo cambiare la composizione del messaggio nella CLI, usando i dati della fonte già accettata. Non occorre spostare il controllo nel trasporto o modificare il protocollo. Nella versione proposta il messaggio contiene già identificatore e revisione: separarli in due righe, una per documento, permette di riconoscere la differenza fra una scelta grafica e una condizione di ammissibilità dei dati. Anche il messaggio atteso dalla verifica della presentazione dovrà seguire il nuovo formato; gli esiti dei rifiuti rimarranno invariati.

Interrompere una chat e lasciare proseguire l'altra

Nella variante del capitolo 6 la selezione sullo schermo non basta a individuare il lavoro da fermare. Conserviamo il controller della chat A e il riferimento al suo tentativo. Invalidiamo soltanto quella generazione e chiediamo la cancellazione soltanto al task corrispondente. Il controller B mantiene la propria identità corrente: una sua risposta resta ammissibile se supera i controlli di chat, generazione e fase.

Una prova trasferibile avvia entrambi i lavori, interrompe A e consegna comunque entrambe le risposte nei due ordini. A deve rifiutare il proprio testo tardivo e restare fermata; B deve completarsi con il proprio testo. Cambiare selezione durante la prova non deve cambiare questi esiti. La selezione e l'arresto sono comandi diversi, anche quando la stessa finestra permette di compierli.

Se cambia invece una fonte nello stesso tentativo, la generazione può rimanere uguale. Il trasporto restituisce identità e revisione effettive; il nucleo le confronta con quelle richieste. Quando l'incarico esige la revisione corrente, un risultato basato sulla vecchia fonte deve essere rifiutato o acquisito di nuovo. Se il mandato ammette una fotografia storica, quella condizione deve essere riconoscibile nella consegna. Il solo riferimento al tentativo non risolve questa scelta.

Per una risposta progressiva il controllo deve accompagnare ogni frammento capace di cambiare testo o fase. Anche la consegna finale e l'errore tardivo devono riferirsi al tentativo ancora ammesso. Controllare soltanto il primo frammento lascia che i successivi attraversino uno stato cambiato. Nel codice isolato il confronto e l'aggiornamento devono rimanere nello stesso tratto senza sospensioni; una nuova attesa richiede di riesaminare le condizioni.

La variante delle letture indipendenti non giustifica un testo mutabile condiviso fra tutti i lavori. Ogni lettura può restituire un valore identificato; il coordinatore compone i valori e il nucleo controlla fonti e revisioni. Se la lettura B dipende dal contenuto di A, prima va risolta quella dipendenza. La concorrenza è adatta quando la relazione fra operazioni la consente.

Salvare una bozza interrotta e riprendere una chiamata

La variante del capitolo 7 richiede prima di rendere riconoscibile lo stato del contenuto. La bozza ricevuta prima dell'interruzione non ha incontrato il completamento del trasporto. Possiamo proporre il salvataggio come bozza interrotta se la persona lo desidera, mantenendo quell'indicazione nel documento e nell'esito. Non possiamo attribuirle la verifica del confronto o presumere che il testo assente fosse irrilevante.

Il salvataggio richiede poi i controlli dell'operazione: contenuto esatto, destinazione ammessa e riscontro della scrittura. La richiesta della persona può autorizzare il salvataggio, ma non trasforma retroattivamente la risposta incompleta in una conclusione del provider. Se il documento deve soddisfare i criteri di un confronto completo, la lacuna va risolta prima di dichiarare la consegna riuscita.

Una chiamata di lettura ricevuta dentro una risposta completa apre invece il confine del tool. Devono essere controllati nome, identificatore, argomenti, risorsa, revisione e accessi. Il risultato deve ritornare con il riferimento alla particolare chiamata. I frammenti mostrati sullo schermo non sostituiscono la proposta completa e il completamento del provider non dimostra l'esecuzione della lettura.

Una terza fonte e un effetto da accertare

Nel capitolo 8 la variante C cambia il contratto del lavoro. Se C è obbligatoria fin dall'inizio, occorre aggiungerla alle fonti richieste e ai criteri della consegna. Se emerge dal contenuto di B, il programma deve conservare il riferimento proposto e risolverlo in un perimetro ammesso, senza permettere che un documento conceda accessi. Il modello può scegliere di approfondire; il componente che possiede la risorsa continua a controllare la lettura. Il massimo dei turni deve consentire il nuovo percorso, ma non decide da solo quali fonti siano necessarie.

Passando al salvataggio, il risultato del tool deve identificare il contenuto scritto, la destinazione e il riscontro autorevole disponibile. Il `call_id` mette in relazione proposta e osservazione nel dialogo; un insieme di identificatori tenuto in memoria non protegge da una ripetizione dopo il riavvio. Serve definire il contratto di idempotenza della risorsa o un modo di ricostruire l'effetto. Se non sappiamo se il file sia già stato scritto, la consegna deve rappresentare quell'incertezza prima di ritentare.

Per l'arresto durante una lettura, il proprietario dello stato invalida il riferimento del tentativo, chi conserva il task chiede la cancellazione e il deposito rifiuta una registrazione riferita alla vecchia generazione. Un effetto già prodotto fuori dal processo richiede comunque un accertamento separato. Nel laboratorio di lettura in memoria possiamo osservare il rifiuto della consegna; non dedurre una capacità di annullare scritture esterne.

Una prova trasferibile deve anche controllare il seguito. Dopo una lettura negata, l'ingresso al modello deve contenere la negazione e il riferimento della chiamata; una nota completa incompatibile deve incontrare un rifiuto del programma. La scelta di

una nota parziale è invece da valutare sul modello reale, senza considerarla dimostrata dalle fixture che l'hanno preparata.

Glossario della raccolta

Agente. Unità alla quale affidiamo un compito e una possibilità di scegliere come proseguirlo. Nel percorso il modello partecipa a quella scelta. Il termine descrive una responsabilità del software e non implica coscienza.

Applicazione agentica. Sistema che collega decisioni, stato, strumenti, controlli e risultati intorno a un obiettivo. Può comprendere un solo agente o partecipanti distinti.

Contesto. Materiale ricevuto dal modello in una determinata richiesta. Può comprendere istruzioni, messaggi e riferimenti; non coincide con tutto ciò che l'applicazione conserva.

Stato del compito. Rappresentazione del lavoro mantenuta dal programma: obiettivo, attività, dati acquisiti, errori, attese ed esiti da verificare.

Esecuzione o run. Tentativo di lavoro identificato dal sistema. Una conversazione può contenerne più di uno; la ripresa deve riconoscere quali dati e azioni gli appartengano.

Tool o strumento. Capacità eseguibile con argomenti, risultati ed errori definiti. Una richiesta del modello deve essere controllata prima di produrre un'operazione.

Skill. Istruzioni e risorse per affrontare un compito. Può orientare il metodo e l'uso degli strumenti; non crea autonomamente nuovi accessi.

Workflow. Processo con attività, condizioni e diramazioni definite dal programma. Può impiegare modelli e attendere eventi.

Mandato. Obiettivo ricevuto con condizioni, risorse e azioni ammesse. Viene mantenuto distinto dalle preferenze ricordate.

Adattatore. Componente che collega un contratto applicativo a un servizio o una risorsa concreta, traducendo richieste ed esiti.

Pacchetto, target e prodotto Swift. Il pacchetto raccoglie sorgenti e manifesto. Un target è un'unità di costruzione con dipendenze proprie; nel percorso Swift il suo codice forma un modulo importabile. Un prodotto espone all'esterno un risultato della costruzione, come una libreria o un eseguibile. Non ogni target interno deve essere un prodotto separato.

Protocollo Swift. Dichiarazione dei requisiti che un tipo conforme deve offrire. Permette al chiamante di usare quel contratto senza dipendere da una sola implementazione concreta. La conformità verificata dal compilatore non dimostra ogni obbligo sul comportamento.

Iniezione delle dipendenze. Scelta di ricevere dall'esterno i collaboratori necessari a un componente. Chi assembla il sistema fornisce un'implementazione conforme; le responsabilità del componente che la usa restano verificabili separatamente.

Modulo e processo. Il modulo organizza il codice e i nomi accessibili. Il processo è un programma in esecuzione che può terminare separatamente da altri programmi. Una libreria caricata dall'applicazione non diventa per questo un processo autonomo.

Broker. Responsabilità applicativa che ammette o nega operazioni secondo i controlli del percorso. Non è necessariamente un processo separato o un altro modello.

Revisione. Identità di una particolare versione. Il suo valore dipende dal modo in cui viene assegnata e collegata al contenuto.

Fixture. Ingresso curato per rappresentare un caso ripetibile. È distinto dagli output prodotti durante la prova.

Diff e commit. Il diff mostra differenze fra versioni. Un commit identifica una revisione nella storia Git. Nessuno dei due, da solo, prova il comportamento in esecuzione.

Idempotenza. Proprietà di un'operazione che, nel contratto e con l'identità previsti, può essere ripetuta senza aggiungere un effetto equivalente indesiderato. Va verificata sul componente che produce l'effetto.

Inferenza. Uso dei parametri già appresi da un modello per elaborare una richiesta e produrre una risposta. Recuperare un appunto modifica il materiale disponibile e non implica aggiornare quei parametri.

Citazione verificabile. Riferimento che permette di ritrovare una fonte e un passaggio nelle condizioni dichiarate. La sua esistenza non dimostra da sola il sostegno della conclusione.

Operazione asincrona, `async` e `await`. Un'operazione asincrona può sospendersi e riprendere. `async` ne dichiara la partecipazione al meccanismo Swift; `await` indica al chiamante un punto di possibile sospensione. Non impongono un nuovo thread o esecuzione parallela.

Task e concorrenza strutturata. Il task rappresenta lavoro asincrono, senza implicare capacità agentiche. Nella concorrenza strutturata le attività figlie, come quelle avviate con `async let`, hanno una durata governata dall'ambito che le contiene. Un task creato con `Task { ... }` richiede invece una gestione esplicita della sua durata nel percorso considerato.

Actor e MainActor. Ambiti di isolamento che coordinano l'accesso allo stato secondo le regole Swift. `MainActor` è condiviso ed è adatto allo stato dell'interfaccia sul percorso principale. L'isolamento protegge gli accessi; non sceglie da solo quale risultato sia ancora pertinente.

Rientranza. Possibilità per altro lavoro ammesso sullo stesso actor di avanzare mentre un'operazione è sospesa. Dopo l'attesa lo stato può essere cambiato e una condizione precedente può non valere più.

Generazione del tentativo. Identità che permette di riconoscere il lavoro ancora ammesso a modificare un particolare stato. Nel laboratorio è un UUID; non è la revisione del documento, un permesso o la versione del modello.

Cancellazione cooperativa. Richiesta di interrompere un task alla quale il lavoro deve reagire. Non garantisce una fine immediata e non annulla effetti esterni già prodotti. Invalidare una generazione permette di rifiutare consegne tardive anche se il servizio non collabora.

Sendable. Protocollo Swift relativo alla possibilità di trasferire valori attraverso confini di concorrenza rispettando i controlli previsti. La conformità non certifica l'attualità dei dati o gli accessi necessari per usarli.

Provider, modello ed endpoint. Il provider offre il servizio; il modello identifica la capacità richiesta; l'endpoint è la destinazione che riceve un determinato tipo di richiesta. Il formato dell'API e le capacità ammesse vanno verificati insieme: cambiare un nome non dimostra la compatibilità.

SSE, Server-Sent Events. Formato per delimitare eventi testuali in un flusso. Nel sottoinsieme studiato, il lettore ricompile righe `data:` e riconosce la fine dell'evento dalla riga vuota. I blocchi ricevuti non coincidono necessariamente con eventi o caratteri completi.

Delta e risposta completa. Il delta è un aggiornamento parziale. Il completamento del provider è un evento distinto e richiede un contenuto valido. Nessuno dei due sostituisce i criteri di completamento dell'incarico.

`call_id`. Identificatore di una particolare chiamata di strumento nel protocollo. Permette di collegare proposta e risultato; non è l'identità della chat, della fonte o una garanzia di idempotenza durevole.

Osservazione dello strumento. Esito identificato di un'operazione: può comprendere un dato acquisito oppure una negazione o un limite. La sua autorità dipende dal componente che lo produce. È materiale per il seguito del ciclo, senza diventare automaticamente un'istruzione da eseguire.

Consegna parziale. Risultato che mantiene riconoscibili le condizioni non soddisfatte e ciò che rimane disponibile. Nel laboratorio richiede fonti mancanti coerenti con gli esiti delle letture; non basta aggiungere l'etichetta «parziale» a una conclusione inventata.

Lecture contestualizzate

Dal prompt al software sviluppa il metodo di lavoro con l'AI: contesto, criteri di accettazione, incrementi, riuso, prove e responsabilità del programmatore. I casi di quel libro sono distinti da quelli di Velia; le pratiche vengono applicate qui al progetto reale.

La documentazione *The Swift Programming Language* è il riferimento primario del linguaggio. La raccolta introduce strutture, protocolli e visibilità quando servono ai problemi; il capitolo 6 aggiunge concorrenza, isolamento e proprietà dello stato. La *documentazione di PackageDescription* permette di approfondire la costruzione del manifesto, distinta dal comportamento dell'assistente.

Le proposte ufficiali Swift [SE-0304](#), [SE-0306](#) e [SE-0316](#) approfondiscono rispettivamente concorrenza strutturata, actor e actor globali. Sono proposte del linguaggio e spiegano i meccanismi usati; non qualificano il comportamento di Velia. SE-0316, di John McCall e Doug Gregor, chiarisce in particolare il modello di isolamento condiviso al quale appartiene `MainActor`.

La documentazione ufficiale OpenAI sulle *risposte progressive*, sulle *chiamate di funzione* e sulle *pratiche di produzione* accompagna i capitoli 7 e 8. Lo *standard SSE del WHATWG* distingue il formato generale dal contratto del servizio. Sono riferimenti dei meccanismi e delle regole: non sono risultati dei laboratori o qualifiche del prodotto.

Le fonti di ricerca e i rapporti dell'episodio OpenAI-Hugging Face sono presentati nell'introduzione con ambito e limiti. Non sono risultati del laboratorio Swift. La guida ai sorgenti distingue quei riferimenti dalle revisioni del codice e dai materiali eseguibili.

Leggere e riprodurre la raccolta di Velia

Questa raccolta comprende introduzione e capitoli 1-8 di *Costruire un'AI agentica, mattone dopo mattone*. Segue l'esperienza di Velia nel progetto Jarvis ed entra nell'architettura Swift. I casi documentali sono inventati per il libro; i componenti richiamati, il manifesto del pacchetto e il cambiamento di ripresa del capitolo 3 provengono dal repository reale.

Il percorso di lettura parte dall'introduzione, prosegue con l'incarico, le tre architetture, il cambiamento del recupero, il primo laboratorio Swift, la mappa dei componenti, concorrenza e stato, il trasporto e il primo ciclo. Le soluzioni e il glossario accompagnano la raccolta. L'ordine costruisce i prerequisiti; non riproduce una cronologia completa dello sviluppo.

Quale codice descrive il libro

Il riferimento generale di Velia è il commit `7d0c7fe19a66982c50bb8e4deef51a3dfcb698b2` del ramo `dev` di Jarvis. Un commit identifica una revisione nella storia Git. Il libro non legge implicitamente il ramo di personalizzazione o la versione dell'app installata sul computer. La documentazione architeturale del 26 ottobre 2026 descrive una fotografia precedente e viene usata come orientamento, confrontandola con i sorgenti del riferimento adottato.

Per l'episodio del capitolo 3 confrontiamo il commit `a44709390ca79ca4a700dea78e2e485a739fa38a` con il suo genitore. Il genitore è la revisione immediatamente precedente nella storia di quel commit. Le istruzioni di progetto e il diff permettono di collegare l'esigenza di interpretazione guidata dal modello alla trasformazione del codice; non forniscono una trascrizione del dialogo di programmazione o risultati di una valutazione del modello.

Responsabilità	Fonte nel repository Jarvis	Uso nella raccolta
Richiesta, contesto e coordinamento	<code>Sources/VeliaDesktop/ChatAgentCoordinator.swift</code> e <code>AssistantController.swift</code> , nella stessa cartella	Confine fra modello e applicazione; candidato, pertinenza e recupero nel capitolo 3.

Responsabilità	Fonte nel repository Jarvis	Uso nella raccolta
Trasporto del modello	ChatModelTransport e OpenAIChatModelTransport, in Sources/VeliaDesktop/ChatAgentCoordinator.swift	Contratto ricevuto dal coordinatore, distinto dal modulo del nucleo.
Esecuzione degli strumenti e deposito	Sources/VeliaCore/AgentRuntime.swift, AgentStore.swift e AgentDatabase.swift, nella stessa cartella	Runtime concreto, accesso allo stato e persistenza SQLite nel capitolo 5.
Esecuzioni e storia del contesto	Sources/VeliaCore/ChatRuns.swift e AgentStore.swift, nella stessa cartella	Distinzione fra conversazione e lavoro registrato.
Taccuino	Sources/VeliaCore/ChatNotebook.swift	Sezioni e basi diverse per appunti, fonti e ipotesi.
Inventario e copertura	Sources/VeliaDesktop/FolderInventory.swift	Distinzione fra trovato, acquisito e restituito, senza attribuirle comprensione semantica.
Moduli e prodotti	Package.swift	Sette target, cinque prodotti e dipendenze rappresentati nel capitolo 5.
Processi ausiliari	Sources/VeliaTerminalHost/main.c e Sources/VeliaPythonHost/main.c	Supervisor in C, distinti dai moduli caricati nel desktop e dal modello.
Conversazioni e tentativi concorrenti	Sources/VeliaDesktop/ChatWorkspace.swift e AssistantController.swift	Controller per chat, selezione, generazione, aggiornamenti e arresto nel capitolo 6.

I manifesti dei listati dei capitoli 7 e 8 identificano sorgenti, righe e impronte dei frammenti. Il manifesto dei listati del capitolo 6 identifica anche le righe dei sorgenti didattici. Gli estratti conservati identificano revisione, sorgente e punto di lettura. Sono frammenti riformattabili nella prosa, non file autonomi compilabili. Il repository di Jarvis completo non è necessario per eseguire il laboratorio di questa raccolta e non viene distribuito implicitamente con esso.

La fonte dell'introduzione precedente è conservata come materiale editoriale. L'introduzione nuova eredita ricordi, episodio, distinzioni e confronto svolto,

aggiornando raccordi e percorso a Velia e Swift. Il riferimento storico alla Centrale operativa conserva la propria funzione di laboratorio comparativo.

Quale programma possiamo eseguire adesso

La raccolta contiene cinque programmi completi, distinti dal prodotto Velia. Il primo è il verificatore delle fonti. Richiede Swift 6 e Python 3 e non usa rete o credenziali. Compila in uno spazio privato, esegue fixture curate e rimuove compilati e cache. Serve a verificare proprietà circoscritte dei riferimenti; non esegue una versione ridotta di Velia e non valuta la comprensione del modello.

Dalla radice del repository:

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-04/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

Il caso frase-non-sostenuta accetta i riferimenti e lascia aperta la valutazione della conclusione. Le verifiche dell'applicazione Velia restano distinte da questo laboratorio.

Il secondo programma è il pacchetto sui confini fra componenti. Richiede anche Swift Package Manager, incluso negli strumenti Swift, e non ha dipendenze esterne. Tre target separano contratti, nucleo e applicazione. Gli adattatori in memoria e da file attraversano lo stesso controllo di identità e revisione; il programma non ripete il controllo delle citazioni del capitolo 4.

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-05/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

Il driver compila una sola volta in una copia privata e confronta gli otto casi con i criteri preparati. Non compila il pacchetto Jarvis e non dimostra la compatibilità della sua distribuzione macOS. Le raccolte precedenti 1-4, 1-5 e 1-6 restano fotografie di lettura; il riferimento corrente comprende i capitoli 7 e 8 e gli apparati aggiornati.

Il terzo programma è l'esperimento sulle risposte tardive. Richiede Swift 6 e Python 3, senza pacchetti esterni. Usa un controller `MainActor` per chat, riferimenti immutabili a chat e generazione e un actor che consegna risposte nell'ordine preparato. Un controesempio separato mostra che il solo isolamento non impedisce una sovrascrittura logicamente superata. Le due letture con `async let` mostrano un ambito strutturato senza misurare il parallelismo.

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-06/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

Gli otto casi controllano due ordini per cambio chat e sostituzione, risposta corrente, arresto, controesempio e letture indipendenti. Non usano ritardi presunti per sincronizzarsi. Il servizio ignora appositamente la cancellazione per provocare la consegna tardiva; questa condizione non è un contratto di trasporto per l'app reale.

Nel sorgente reale `ChatWorkspace.selectLoaded` conserva il controller della chat lasciata; i collegamenti creati da `ensureCollaborationController` acquisiscono l'identità della chat. `AssistantController.append`, il percorso finale, quello d'errore e `cancelResponse` permettono di seguire il controllo della generazione. `AgentRuntime` e `AgentStore.finishStep` permettono di distinguere attese, ammissione del completamento e registrazione dei consumi, che può avvenire anche dopo l'arresto. Gli estratti identificati nel manifesto conservano il contesto di questi passaggi; non sono file autonomi da compilare.

Richiesta progressiva e ciclo dei capitoli 7 e 8

Il quarto programma è il lettore del flusso: costruisce il corpo essenziale della richiesta e riceve byte SSE curati attraverso un trasporto preparato. I tredici casi controllano frammentazione, identità, completamento, rifiuti, errori e limiti. Non apre HTTP, non riceve credenziali e non dimostra la compatibilità con un account. Il quinto è il ciclo, che riusa quel lettore e restituisce osservazioni identificate al turno successivo. Le nove sequenze controllano letture, accessi negati, contratto, duplicati e arresto per limite; comprendono una conclusione semanticamente falsa che passa i soli controlli dei riferimenti.

```
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-07/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py run -- \
  python3 ai-agentica/esempi/velia/capitolo-08/esegui.py \
  --verifica
python3 quality-gates/storage/gate.py check
```

Il lettore reale è `ChatAgentStreamParser` in `Sources/VeliaDesktop/ChatAgentCoordinator.swift`. Il metodo `OpenAIChatModelTransport.complete` nello stesso file descrive `URLSession`, intestazioni, controlli HTTP e cancellazione; `OpenAIRedirectPolicy` in `OpenAIClient.swift` rifiuta i redirect. `KeychainTokenStore`, nello stesso file del client, e

`SystemKeychainStorage` in `TouchIDCredentials.swift` identificano la responsabilità delle credenziali. Il libro non accede ai valori memorizzati.

Per il ciclo si leggono in `ChatAgentCoordinator.respondWithMemory` la costruzione del corpo, l'esecuzione di `chat_model`, l'output completo, l'identità delle chiamate e l'aggiunta di `function_call_output`. La transizione a completato o parziale è applicativa. `AgentRuntime.execute` risolve capacità e schema prima del passo; deposito e handler applicano i controlli dei rispettivi confini. Il ciclo didattico ha stato in memoria e massimo tre turni: non trasferisce questi limiti al prodotto.

Le fonti ufficiali consultate il 28 ottobre 2026 sono le guide OpenAI **streaming**, **function calling**, **stato della conversazione** e **pratiche di produzione**, oltre allo **standard SSE del WHATWG**. Descrivono contratti e formato, distinti dalla fotografia Git e dalle prove del programma. Le fixture sono ingressi curati; nessuna registrazione di prove o di un servizio viene conservata nella raccolta.