Università degli Studi di Napoli Federico II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Corso di Laurea in Informatica

Sperimentazione di una tecnica per la correzione automatica di difetti causanti NPE nel dataset Defects4J

Relatore:
Prof. Porfirio Tramontana

Candidato:
Mario Ponticiello
Matricola N86004563

Anno Accademico 2025/2026

Indice

Capitolo 1

Introduzione e definizione del problema

1.1 Contesto: Il costo della manutenzione del software e il ruolo dell'*Automated Program Repair*

I difetti del software rappresentano una delle sfide principali e più costose nell'intero ciclo di vita dello sviluppo informatico, comportando un impiego massiccio di tempo, denaro e risorse umane [24].

Come evidenziato da diversi studi, il processo di risoluzione dei bug è un'operazione notoriamente ardua, laboriosa e dispendiosa in termini di tempo, la quale arriva a dominare le agende degli sviluppatori e a costituire la voce di costo predominante nella manutenzione del software [24].

L'impatto economico derivante dalla scarsa qualità del codice è di proporzioni critiche: secondo un recente rapporto del Consortium for Information & Software Quality (CISQ), citato nelle ricerche di Huang et al., nel solo 2020 gli Stati Uniti hanno sostenuto costi pari a circa 2,08 trilioni di dollari a causa di software di bassa qualità, di cui ben 1,56 trilioni imputabili direttamente ai fallimenti operativi dei sistemi (*Cost of Poor Software Quality*).

Tali cifre dimostrano come gli ingegneri del software debbano investire una quantità considerevole di sforzi per garantire il corretto funzionamento dei programmi, riducendo i difetti attraverso un continuo e tedioso processo manuale che include l'identificazione dell'errore e la sua successiva risoluzione [16].

Tuttavia, questo approccio tradizionale, basato esclusivamente sull'intervento umano per l'intero ciclo di debugging e riparazione, sta diventando rapidamente insostenibile. La società moderna dipende sempre di più dai sistemi software, il cui rapido sviluppo e la crescente popolarità su larga scala ne hanno aumentato esponenzialmente la complessità strutturale [17]. Dikici e Bilgin sottolineano come la crescente complessità delle applicazioni moderne renda le metodologie di ispezione umana inadeguate alle attuali esigenze di manutenzione.

I difetti software, che derivano per lo più da problemi storici radicati nelle fasi iniziali di progettazione, rendono la riparazione manuale un compito sempre più prono all'errore e incompatibile con i cicli di sviluppo rapidi odierni [17, 11]. Di conseguenza, emerge la necessità impellente di individuare soluzioni scalabili che possano ridurre in modo drastico il carico di lavoro manuale [24].

In questo scenario critico, l'*Automated Program Repair* (APR) si inserisce come una soluzione trasformativa mirata a risolvere direttamente l'insostenibilità dei costi e dello sforzo umano appena descritti [11]. Come suggerito da Huang et al., l'obiettivo principale dell'APR è quello di trasferire il pesante lavoro di manutenzione manuale verso un processo di riparazione automatizzato, preciso ed efficiente.

Sfruttando avanzate tecniche computazionali, che spaziano dalla tradizionale ricerca euristica (*search-based*) ai metodi basati su vincoli e *template*, fino ad arrivare ai più recenti approcci guidati dai dati tramite apprendimento automatico e profondo (*deep learning*), i sistemi APR sono specificamente progettati per analizzare il codice e rilevare le anomalie in autonomia [17, 11].

Il loro scopo ultimo è quello di generare e implementare *patch* valide con un intervento umano minimo o del tutto assente [11].

Automatizzando il processo del *bug fixing*, le tecnologie APR si posizionano come una risorsa cruciale: non solo accelerano i cicli di sviluppo riducendo sensibilmente le tempistiche e i costi necessari per risolvere i difetti, ma contribuiscono in modo determinante alla creazione di sistemi software più robusti, affidabili e di alta qualità [11, 24].

1.2 Definizione del Problema

1.2.1 La complessità delle NullPointerException

Le NullPointerException (NPE) rappresentano il focus centrale di questa tesi in quanto costituiscono la causa principale di eccezioni non gestite che determinano il crash delle applicazioni in ambienti di produzione [13]. Rispondendo alle frequenti interrogazioni sulla criticità di tale difetto, la letteratura empirica dimostra che le NPE non sono un problema marginale, bensì il bug di gestione della memoria in assoluto più comune: esse costituiscono il 37,2% degli errori in sistemi complessi come Mozilla e Apache, causano oltre il 40% delle eccezioni in ambiente Android e sono responsabili fino al 70% dei fallimenti nelle applicazioni Java in produzione [13, 47]. Oltre alla loro elevata frequenza, un singolo puntatore nullo possiede la criticità intrinseca necessaria per causare il crash di un'intera applicazione [13]. Pertanto, le NPE vengono qui affrontate poiché costituiscono un *representative bug type* che si rivela estremamente difficile da correggere in modo automatizzato [47]. A differenza di altri difetti di natura puramente semantica, che possono corrompere lo stato del programma, le NPE presentano il vantaggio diagnostico di rivelarsi palesemente attraverso l'interruzione improvvisa del programma, il crash. Questa manifestazione esplicita è di fondamentale importanza per gli attuali strumenti di Automated Program Repair. L'intero processo di riparazione automatica dipende infatti dalla presenza di suite di test preesistenti ed efficaci: i tool APR necessitano in input di almeno un caso di test fallito che funga da oracolo, il quale riproduca l'eccezione non gestita e inneschi il crash [13, 47]. Di conseguenza, il dominio di questo studio si focalizza precisamente sulla riparazione automatica di

NullPointerException rivelate dall'esecuzione di casi di test in input, le quali si manifestano sotto forma di eccezioni a runtime che interrompono la corretta esecuzione del programma sotto test.

In ottica ingegneristica, è fondamentale chiarire che le NPE non costituiscono banali errori sintattici, ma insidiosi bug legati alla gestione dei puntatori. La loro insorgenza e il loro comportamento sono per natura dipendenti dal flusso di esecuzione (control-flow) e dal flusso dei valori (value-flow). Come evidenziato dalla letteratura, l'analisi statica limitata al codice locale è insufficiente per avere un quadro completo delle variabili compatibili; per comprenderne appieno la natura è necessaria un'analisi (spesso dinamica o guidata dal flusso di valori) del contesto a tempo di esecuzione [13, 47].

La difficoltà principale nel debugging delle NPE risiede in un marcato disallineamento spaziale e logico tra il difetto originario e il punto in cui l'eccezione viene lanciata. Il punto esatto in cui il programma va materialmente in crash rappresenta tipicamente solo la fine di una complessa catena di propagazione originata altrove. Un difetto originario, che nella maggior parte dei casi consiste in una mancata inizializzazione di un oggetto o in un controllo di nullità assente, risiede spesso a notevole distanza a livello di codice [47]. Di conseguenza, una NPE e la sua effettiva posizione di correzione (bug-fixing location) possono estendersi in modo inter-procedurale attraverso molteplici funzioni all'interno di codebase di grandi dimensioni [47].

Questa frammentazione strutturale pone sfide profonde per gli strumenti di *Automated Program Repair*. Ignorando le complesse dipendenze di controllo e dei dati (il value-flow inter-procedurale), i tradizionali approcci APR operano su spazi di ricerca immensi generando spesso riparazioni localizzate esclusivamente al punto di crash [47]. Tali approcci limitati (come i *single-line edits* per saltare una singola istruzione o inserire un controllo banale) portano frequentemente alla generazione di patch "plausibili ma errate" (*plausible but incorrect patches*): soluzioni che riescono a superare la suite di test ma introducono comportamenti semanticamente indefiniti nel prosieguo del programma [47]. Risolvere in modo definitivo questi bug complessi non richiede un semplice mascheramento dell'eccezione, ma necessita di modifiche strutturate (*multi-point edits*) capaci di ripristinare il corretto flusso dei dati fin dalla sua origine, evidenziando così i limiti degli strumenti di riparazione basati unicamente su template sintattici limitati o approcci di ricerca ciechi [47].

1.2.2 I limiti degli LLM nell'Automated Program Repair

Negli ultimi anni, i *Large Language Models* (LLM) si sono affermati come lo standard de facto per automatizzare *task* complessi di ingegneria del software, incluse le più recenti tecniche di *Automated Program Repair* a livello di *repository* [53, 29]. Tuttavia, nonostante le notevoli capacità dimostrate nella risoluzione di frammenti di codice isolati, un'analisi accademica rigorosa rivela che i sistemi basati su LLM "puri" presentano difetti strutturali quando sono chiamati ad affrontare *bug* complessi e semanticamente densi. Con il termine LLM "puro" si intende l'utilizzo del modello di base che, pur elaborando le informazioni locali fornite nel *prompt* (come la descrizione testuale del problema e le righe di codice immediatamente sospette), opera in totale isolamento rispetto all'ecosistema dell'intero progetto. A causa dei rigidi limiti sui token, infatti, è impossibile e impraticabile includere l'intero contesto del *repository* all'interno del *prompt*; di conseguenza, senza l'ausilio di infrastrutture esterne, il modello è costretto

ad affidarsi esclusivamente alle proprie congetture statistiche derivate dall'addestramento, ignorando le reali relazioni tra i file. Tali limiti emergono in modo evidente di fronte a problematiche che richiedono una profonda comprensione del flusso di controllo, dei dati o delle dipendenze di stato. Come sottolineato dalla letteratura, la debolezza dei modelli non risiede in una specifica categoria di eccezione, bensì nell'incapacità di gestire difetti che richiedono coerenza semantica globale. Ignorando le catene di chiamate transitive e le architetture distribuite su più file, gli LLM incorrono sistematicamente in quelle che vengono definite "allucinazioni logiche" e "schematiche" [52]. Il codice generato in risposta al *prompt* locale può quindi apparire plausibile e superare la compilazione, ma fallisce nel risolvere il *bug* reale introducendo errori profondi: iterazioni errate su collezioni, omissione di mutazioni di stato necessarie, chiamate ad API inesistenti, o gravi discrepanze di tipo e firma [52, 53]

Il motivo tecnico alla base di questi difetti strutturali risiede nel meccanismo fondamentale di funzionamento dei modelli linguistici. Come rilevato da Zhang et al., gli LLM generano il codice operando primariamente sulla base della probabilità statistica, ovvero prevedendo ricorsivamente un *token* dopo l'altro. Questo processo avviene senza alcun meccanismo connaturale capace di garantire il rispetto dei requisiti formali o semantici del linguaggio di programmazione [52]. Precisano, inoltre, che il vantaggio principale degli LLM, ovvero l'eccellente capacità di riconoscimento dei *pattern*, costituisce al contempo la loro più grande limitazione: i modelli tendono a generare codice appoggiandosi esclusivamente ai *pattern* comuni osservati nei dati di addestramento, piuttosto che derivare una comprensione profonda delle specifiche intenzioni e necessità logiche del problema [53]. Questo approccio puramente predittivo conduce inesorabilmente al fenomeno delle allucinazioni [52, 53]. In ambito di generazione del codice, le allucinazioni si manifestano tipicamente quando il modello produce *patch* che appaiono sintatticamente corrette e del tutto plausibili, spesso riuscendo persino a superare la fase di compilazione senza errori, ma che si rivelano fondamentalmente errate o inconsistenti dal punto di vista della logica semantica. Tali aberrazioni si dividono in errori logici (come l'iterazione su collezioni errate o l'omissione di mutazioni di stato vitali) ed errori schematici (che includono violazioni di firme, discrepanze di tipo e chiamate a funzioni mai definite) [52].

Questo limite statistico e predittivo si traduce direttamente nel fenomeno dell'*overfitting*, ampiamente documentato da Pabba et al. Gli studi dimostrano come i sistemi basati esclusivamente su LLM tendano a iper-localizzarsi sulle righe di codice immediatamente sospette, tentando di ripararle in totale isolamento rispetto all'intero ecosistema del software. Il risultato è la generazione di *patch* che soffrono di un marcato *overfitting*: il codice prodotto riesce a risolvere gli specifici esempi falliti forniti nella descrizione del *bug*, riuscendo magari a superare la *test suite* locale, ma si rivela incapace di generalizzare. La riparazione si limita quindi ad agire come un mero cerotto superficiale che fallisce nel gestire i casi limite (*edge cases*) e non soddisfa i requisiti di coerenza più ampi e profondi dell'applicazione [29].

Infine, questa tendenza all'*overfitting* e alla generazione di soluzioni localmente ottime ma globalmente fallate è legata alla mancanza di una reale consapevolezza del codice da parte dei modelli. Zhang et al. sottolineano come l'assenza di un contesto a livello di intero progetto (*repository-level context*) impedisca agli LLM puri di comprendere le dipendenze profonde necessarie per vere riparazioni [53]. In scenari di sviluppo reali, la stragrande maggioranza delle funzioni dipende da entità definite altrove nel

progetto o da API personalizzate; tuttavia, i rigidi limiti sui *token* dei modelli rendono impossibile e impraticabile l'inclusione dell'intero contesto del *repository* nel *prompt*. Di conseguenza, ignorando le catene di chiamate transitive e i vincoli architetturali distribuiti su molteplici file, i modelli "puri" si affidano a supposizioni statistiche cieche, generando conflitti di dipendenza e riparazioni semanticamente vuote che non possono integrarsi in sicurezza in basi di codice complesse [53, 52].

1.2.3 L'illusione della Perfect Fault Localization

Le elevate percentuali di successo vantate dalla recente letteratura sui *tool* di *Automated Program Repair* basati su *Large Language Models* sono spesso sovrastimate a causa dell' utilizzo della *Perfect Fault Localization* (PFL) [25], Gran parte della ricerca accademica valuta l'efficacia dei modelli fornendo loro l'esatta riga di codice o la specifica funzione contenente il difetto, assumendo artificialmente che tale informazione sia sempre disponibile e accurata [25]. Di conseguenza, la letteratura fornisce misure della capacità di generare riparazioni valide che sono di fatto rigorosamente limitate a casi di studio nei quali la posizione del difetto era già precedentemente nota e comunicata a priori al modello [25, 8]. Tuttavia, questo scenario è del tutto irrealistico nella pratica industriale, creando una percezione distorta delle reali potenzialità di questi strumenti, i quali finiscono per essere testati in un ambiente idealizzato e non replicabile sul campo [8].

Nella realtà operativa, infatti, la fase di identificazione del *bug* non è un dato di partenza garantito. È fondamentale distinguere gli strumenti a disposizione: se da un lato gli sviluppatori utilizzano gli oracoli (come le *suite* di *test*) per riconoscere il fallimento o il successo di un programma, dall'altro lato, per localizzare effettivamente la posizione del difetto all'interno del codice sorgente, non dispongono di puntatori infallibili. Essi devono pertanto affidarsi necessariamente a tecniche di *Automated Fault Localization* (AFL), le quali sono per natura imperfette [25]. La contrapposizione tra gli scenari di PFL e AFL svela una fragilità degli attuali sistemi di riparazione. Il lavoro di [Campos et al., 2025] evidenzia esplicitamente un calo significativo dell'accuratezza quando i *tool* vengono testati in condizioni realistiche e imperfette. Nello specifico, lo studio dimostra che l'effettiva posizione del *bug* compare tra i primi tre candidati identificati dallo strumento di localizzazione solamente in 28 casi su 100, portando inevitabilmente i modelli linguistici a operare su porzioni di codice irrilevanti e abbattendo le metriche di successo. L'abbandono della localizzazione perfetta comporta crolli prestazionali drastici, che spaziano dal -7,83% per Gemini 2.0 Flash fino al -16,20% per Claude 3.7 Sonnet [8]. Questa profonda discrepanza viene ulteriormente corroborata dai dati quantitativi forniti da [Li et al., 2024]. Valutando il sistema GIANTREPAIR sotto la rassicurante assunzione della *Perfect Fault Localization*, il *tool* è stato in grado di riparare con successo ben 171 *bug*. Quando però lo stesso strumento è stato messo alla prova nello scenario ben più sfidante e realistico dell'*Automated Fault Localization*, il numero di *bug* risolti è crollato drasticamente a soli 64 [25]. Questo divario incolmabile dimostra che queste *performance* eccellenti dipendono quasi esclusivamente dall'inserimento artificiale della localizzazione esatta.

La causa sottostante a un fallimento così clamoroso dei modelli in assenza di PFL risiede nella loro grave carenza di reali capacità di ragionamento, mascherata da un'eccessiva memorizzazione del codice sorgente. Il rigoroso impianto analitico basato sul test

delle ipotesi introdotto da [Kong et al., 2025a] smaschera l'illusione della comprensione semantica degli LLM, permettendo di distinguere analiticamente la vera risoluzione di un problema dalla riproduzione dei dati di addestramento. [23]

L'esperimento dimostra questa dinamica testando LLM (GPT-3.5 e CodeLlama) su due *dataset* distinti: Defects4J, storico e presumibilmente memorizzato, e GitBug-Java, recente e sconosciuto ai modelli. Le statistiche presentate nello studio sono inequivocabili: nel noto *dataset* Defects4J, una percentuale altissima delle *patch* ritenute corrette, nello specifico il 78,83% per GPT-3.5 e l'87,42% per CodeLlama-7b, non è in alcun modo frutto di ragionamento o reale riparazione, ma consiste in un "*exact match*", ossia in una rigurgitazione letterale di codice memorizzato durante la fase di *training*.

Ancora più allarmante è la prova fornita da questi esperimenti attraverso l'alterazione sistematica del codice. I ricercatori hanno mutato pesantemente il codice affetto da *bug*, omettendo il contesto logico essenziale o scambiando e rinominando le variabili chiave. Se il modello stesse realmente ragionando, dovrebbe adattare la sua correzione alle nuove variabili. Invece, persino quando il codice viene significativamente mutato, omettendo o alterando il contesto per cui le correzioni originali non avrebbero più validità semantica, l'81,82% delle *patch* di GPT-3.5 e l'88,24% di quelle di CodeLlama-7b continuano in modo implacabile a essere identiche alle soluzioni originariamente memorizzate, richiamando variabili ormai inesistenti nel *prompt*.

In conclusione, i dati dimostrano che, privati del "suggerimento" esatto e del contesto inequivocabile garantito dalla PFL, i modelli linguistici non sono in grado di orientarsi autonomamente; sprovvisti di una reale capacità di *problem-solving*, essi si limitano a restituire *pattern* memorizzati, fallendo irrimediabilmente nella risoluzione dei *bug* non appena lo scenario si discosta dai confini sicuri dei test di laboratorio [23].

1.2.4 La necessità di una guida semantica

Le evidenze raccolte nelle sezioni precedenti convergono verso un unico nodo critico: né gli approcci APR tradizionali né gli LLM privi di una localizzazione formale riescono ad affrontare la natura inter-procedurale delle NPE, poiché entrambi operano su un contesto semantico insufficiente. Superare questo ostacolo impone un radicale cambio di paradigma.

la straordinaria capacità generativa e la flessibilità dei *Large Language Models* devono essere necessariamente accoppiate a un "motore di ragionamento" esterno, formale e rigoroso. È essenziale promuovere una profonda integrazione tra intelligenza artificiale e le tecniche tradizionali e formali di analisi del programma. Autori recenti supportano con forza questa necessità di ibridazione; ad esempio, [Feng et al., 2025] dimostrano che l'unione tra il rigore strutturale dell'analisi statica e le capacità semantiche degli LLM è la chiave per colmare la "*semantic sparsity*" e abbattere le allucinazioni intrinseche dei modelli [14]. In parallelo, [Abtahi e Azim, 2024] evidenziano l'importanza di potenziare gli LLM attraverso l'analisi statica del codice proprio per prevenire la generazione di *output* apparentemente plausibili ma logicamente scorretti [1]. Infine, [Sepidband et al., 2026] confermano che vincolare l'attenzione del modello con un ragionamento causale mirato risulta assolutamente essenziale per convertire il recupero di frammenti di codice isolati in una riparazione efficace [34].

A tal fine, questa ricerca si baserà sull'integrazione di *LogicFL* [22], uno strumento di localizzazione dei guasti specificamente progettato per le *NullPointerException* (NPE). A differenza degli approcci basati unicamente su euristiche probabilistiche, *LogicFL* si fonda sulla programmazione logica (Prolog) per ricostruire in modo tracciabile e privo di ambiguità il percorso del valore *null* fino alla sua origine esatta. È proprio questa rigorosa tracciabilità causale a motivarne l'impiego come "guida" semantica per i modelli generativi: il funzionamento dello strumento e il modo in cui esso viene integrato nell'architettura proposta saranno descritti nel dettaglio nel capitolo successivo.

1.2.5 Contributi della tesi

Il problema affrontato in questa tesi è la riparazione automatica di *bug* di tipo *NullPointerException* in programmi Java reali *senza* ricorrere alla *Perfect Fault Localization* (PFL). Come argomentato nelle sezioni precedenti, i sistemi APR allo stato dell'arte raggiungono percentuali di successo elevate in larga parte perché *assumono di sapere già dove correggere*: ricevono in ingresso la posizione esatta del difetto, un'informazione che nella pratica industriale non è però mai realmente disponibile. HybridRepair rimuove questa assunzione: non riceve la posizione del difetto, ma deve individuarla autonomamente a partire dal solo identificatore del *bug*.

Il contributo principale è **HybridRepair**, un sistema neuro-simbolico che integra la localizzazione causale formale di *LogicFL* [22] basata su inferenza logica in Prolog, con la generazione di *patch* affidata a GPT-4o, operando in modalità *fully-automated* su 37 *bug* NPE estratti dal *benchmark* standard Defects4J v2, distribuiti su 12 progetti Java *open source*. In questo schema *LogicFL* localizza il guasto in modo **autonomo**, mentre GPT-4o genera la correzione guidata dalla **diagnosi causale** prodotta a monte.

Sul piano sperimentale, HybridRepair genera *patch* plausibili per **22 bug su 37 (59,5%)**; di queste, **15 sono valide** (68,2% di precisione), corrispondenti al **40,5% del dataset complessivo**. Confrontato con VibeRepair [57], sistema allo stato dell'arte che opera però sotto l'assunzione di *Perfect Fault Localization*, HybridRepair supera il *baseline* in termini di *patch* valide assolute: **40,5% (15/37) contro 37,8% (14/37)**. Il dato decisivo non è tuttavia il margine numerico, ma le condizioni in cui è ottenuto: mentre VibeRepair richiede una localizzazione esterna del difetto che nella realtà non è mai disponibile, HybridRepair, grazie all'integrazione di LogicFl [22], la produce autonomamente, risultando così direttamente applicabile in un contesto di produzione.

Capitolo 2

Approccio e strumenti

2.1 L'idea: unire logica e intelligenza artificiale

Come abbiamo visto nel capitolo precedente, il vero limite dei moderni sistemi di riparazione automatica del codice non è la mancanza di "creatività" nel generare soluzioni, ma la mancanza di una **guida rigorosa**. Quando lasciamo che un Large Language Model lavori da solo, senza vincoli precisi, finisce per inventare parti di codice inesistenti (allucinazioni) o per produrre soluzioni che funzionano solo in apparenza (overfitting). All'estremo opposto, gli strumenti puramente logici e formali sono estremamente precisi nell'individuare *dove* si trova l'errore, ma non hanno la flessibilità necessaria per scrivere il codice che lo risolve in un contesto reale.

L'idea alla base di **HybridRepair** nasce proprio da qui: il mondo logico/simbolico e quello neurale/LLM non sono in competizione, ma **complementari**. per la diagnosi si impiega **LogicFL**, uno strumento preesistente, ma il contributo di questo lavoro è la progettazione dell'**architettura di integrazione** che trasforma il suo referto formale in una guida strutturata per la sintesi LLM, con una validazione che va oltre i semplici test automatici.

2.1.1 La filosofia: separare diagnosi e sintesi

Il principio fondamentale di HybridRepair è una netta **separazione dei compiti**:

> **L'analisi logica si occupa di trovare la causa del problema (diagnosi) mentre l'LLM si occupa di scrivere il codice che lo risolve (sintesi).**

La diagnosi. Capire da dove arriva un valore `null`, seguire il percorso di un errore attraverso più funzioni e individuare il punto esatto in cui il programma va in crash richiede un rigore logico e deterministico. Sono garanzie che solo la programmazione logica (come Prolog) può offrire.

La sintesi. Una volta che sappiamo con certezza *cosa* correggere, il problema diventa *come* scriverlo in codice Java pulito, che rispetti lo stile del progetto e superi i test automatici. Qui gli LLM sono insuperabili: comprendono il contesto e generano codice flessibile con grande efficacia.

2.1.2 L'approccio ibrido in pratica: l'esempio Csv-4

Per chiarire il funzionamento, analizziamo un caso reale: il bug **Csv-4**, un errore di tipo `NullPointerException` (NPE) tratto dal benchmark Defects4J e situato all'interno di Apache Commons CSV, nella classe `CSVParser`.

Questo esempio è interessante per un motivo controintuitivo: è un caso in cui *sia* un sistema basato solo su LLM *sia* l'approccio ibrido arrivano alla patch corretta. Proprio per questo è utile come esempio motivazionale: ci permette di spostare l'attenzione dal banale "chi supera i test" al punto che davvero conta, cioè **con quali garanzie** e attraverso quale processo si arriva a una soluzione che non solo "sembra giusta", ma *è* giusta. Come vedremo, è esattamente qui che emerge il contributo distintivo dell'approccio ibrido: non tanto nel *generare* la correzione, cosa che oggi un buon LLM sa fare, quanto nel **produrre da se la diagnosi** che rende quella correzione più affidabile. È la concretizzazione del principio "separare diagnosi e sintesi" enunciato sopra.

Il problema

Il metodo `getHeaderMap()` restituisce una *copia* della mappa che associa i nomi delle colonne alla loro posizione. Il suo Javadoc, però, si limita a dire "returns a copy of the header map": non specifica cosa accada quando un header non è definito. Quella parte del contratto non è documentata: è fissata dall'*oracolo di test* `testNoHeaderMap()`, che con un `assertNull` richiede che, in assenza di header, il metodo restituisca `null` e non una mappa vuota. È una specifica *comportamentale*, confermata anche dalla correzione applicata dagli sviluppatori di Apache Commons CSV, secondo cui `null` ("header assente") e {} ("header presente, zero colonne") non sono intercambiabili.

Nel codice originale, quando manca l'header, il campo `this.headerMap` resta `null`; alla riga 288 il metodo tenta comunque di costruirne una copia (`new LinkedHashMap<>(null)`) e il risultato è il crash:

```
public Map<String, Integer> getHeaderMap() {
    return new LinkedHashMap<String, Integer>(this.headerMap);
       // NPE se headerMap e' null
}
```

Il `null` non nasce qui: nasce nel metodo `initializeHeader()`, dove la variabile `hdrMap` viene inizializzata a `null` e restituita immutata quando non c'è un header; da lì viene assegnata al campo nel costruttore e infine provoca il crash. È una catena causale che attraversa più metodi.

La trappola: una patch che "sembra giusta"

C'è una correzione che viene immediatamente in mente e che sembra perfetta: se manca l'header, restituiamo una **mappa vuota**.

```
public Map<String, Integer> getHeaderMap() {
    // "Niente header? Restituiamo una mappa vuota": il crash
       sparisce...
    return this.headerMap == null ? new LinkedHashMap<>() : new
        LinkedHashMap<>(this.headerMap);
}
```

Compila, elimina l'NPE e a prima vista è del tutto ragionevole. In realtà **viola il contratto**: l'API deve restituire `null`, non {}. Il test lo smaschera subito:

```
java.lang.AssertionError: expected null, but was:<{}>
    at CSVParserTest.testNoHeaderMap(CSVParserTest.java:670)
```

Questa è la classica insidia di un approccio puramente "code-centric": una patch che toglie il sintomo (il crash) ma cambia in silenzio il significato del metodo.[1] Il punto chiave è che, per evitarla, non serve un'IA "più intelligente": serve che il sistema sappia con precisione *qual è il comportamento atteso*.

Come la affronta VibeRepair (l'IA che ragiona sulla specifica)

VibeRepair [57] è uno dei migliori sistemi APR su Defects4J basati su LLM del 2026. Non è un LLM "ingenuo" che riscrive il codice in un colpo solo: è un approccio **specification-centric**. Invece di modificare direttamente il codice, traduce il metodo difettoso in una *specifica del comportamento*, ne inferisce l'intento corretto, ripara la specifica e solo allora rigenera il codice; inoltre usa il feedback dei test falliti per raffinare la propria ipotesi. Su Csv-4 questo processo lo porta alla patch corretta:

```
public Map<String, Integer> getHeaderMap() {
    if (this.headerMap == null) {
        return null;
    }
    return new LinkedHashMap<String, Integer>(this.headerMap);
}
```

VibeRepair, quindi, **non cade nella trappola**: evita sia il crash sia la mappa vuota. Vale però la pena notare *su cosa* poggia questo successo. Primo: come la maggior parte dei sistemi LLM su Defects4J, VibeRepair riceve già la zona difettosa su cui lavorare (la classica assunzione di "fault localization perfetta"): non individua da sé il guasto. Secondo: la sua diagnosi è una specifica *inferita* dal solo metodo difettoso e raffinata sul feedback dei test falliti, non dalla conoscenza dell'intero codice, un'astrazione che può catturare bene l'intento, come in questo caso, oppure mancarlo. La correttezza, in altre parole, dipende dalla *bontà di un'inferenza*, non da una garanzia formale.

Come la affronta HybridRepair (diagnosi formale + sintesi guidata)

L'approccio ibrido raggiunge la stessa patch corretta, ma per una strada diversa, in tre passi:

1. **Step 1 (LogicFL, diagnosi).** È il passo che fa la differenza. Senza ricevere alcun suggerimento su *dove* si trovi il bug, il motore logico esegue il test fallito e ricostruisce *autonomamente* l'intera catena dell'errore (non solo la riga del crash):

   ```
   npe(line(288), null_expr(headerMap)).
   caused_by(npe(line(288)), null_literal(line(325))).      %
      Map<...> hdrMap = null;
   caused_by(npe(line(288)), return(line(351), hdrMap)).    %
      initializeHeader() ritorna hdrMap
   ```

[1] Non è un esempio costruito a tavolino: la patch alla mappa vuota è esattamente la *prima* soluzione proposta dall'LLM all'interno di HybridRepair, prima che il loop di validazione la respingesse (cfr. Step 3).

```
caused_by(npe(line(288)), assign(line(244), headerMap)).%
    this.headerMap = initializeHeader()
```

Questa catena non è lo stack trace. Lo stack trace dell'NPE contiene solo il punto di crash, la riga 288, più due frame interni della JDK (`LinkedHashMap.<init>`) e il test chiamante, ma *non* le righe 325, 351 e 244: quelle vengono eseguite durante la costruzione dell'oggetto, sono già ritornate quando l'eccezione scatta e perciò non figurano sulla pila. Lo stack trace dice *dove* il programma è morto; la catena di LogicFL dice *da dove arriva il* ***`null`*** e come si propaga, è un'analisi di flusso dei dati.

Quindi all'LLM non viene detto genericamente "c'è un crash da qualche parte". Riceve un referto preciso: *il* ***`null`*** *nasce alla riga 325, si propaga attraverso le righe 351 e 244, e provoca il crash alla riga 288.* È esattamente il passo che un sistema puramente LLM **non compie**: dove VibeRepair dà per scontato di sapere già quale metodo toccare, qui la localizzazione del guasto e la sua causa non sono un'assunzione di partenza, ma un **risultato dimostrato** dal motore logico.

2. **Step 2 (Estrazione AST, contesto).**
   Il sistema isola le sole ∼35 righe dei due metodi rilevanti (cioè `getHeaderMap` e `initializeHeader`), scartando le altre ∼460 righe del file. In questo modo l'LLM ha una visuale pulita e mirata sul punto da correggere.

3. **Step 3 (Sintesi LLM, con paletti e validazione).**
   L'LLM riceve la causa esatta, un contesto pulito e una regola esplicita nel prompt: *non far sparire il crash con un oggetto fittizio; superare i test non basta.* Ciononostante, la sua prima proposta restituisce di fatto una mappa vuota; è il loop di validazione, rieseguendo la suite, a respingerla con `expected null, but was:<{}>`. È il test a fissare il comportamento atteso; l'LLM legge la traccia e corregge in `null`:

   ```
   public Map<String, Integer> getHeaderMap() {
       return this.headerMap == null ? null : new
           LinkedHashMap<>(this.headerMap);
   }
   ```

   Tutti i 48 test passano e il controllo semantico conferma che il contratto è rispettato.

Questa patch ottiene esattamente lo stesso risultato della correzione applicata nella realtà dagli sviluppatori di Apache Commons CSV. La differenza, in definitiva, sta negli **input**: VibeRepair esige che il guasto sia *già localizzato*; HybridRepair parte dagli stessi test e dallo stesso codice, mentre la localizzazione, tramite la catena causale, la *produce* da solo. Richiede cioè meno informazioni in ingresso e ne restituisce di più, per dimostrazione, non per inferenza, e in modo riproducibile.

Il confronto

Tabella 2.1: Confronto tra VibeRepair e HybridRepair sul bug Csv-4

Caratteristica	**VibeRepair (Solo IA)**	**HybridRepair (Logica + IA)**
Input richiesti	Codice eseguibile + test falliti + *posizione del guasto* (data)	Codice eseguibile + test fallito
Localizzazione del guasto	Input (assunta nota: FL perfetta)	Output (calcolata da LogicFL)
Causa dell'errore	Inferita come specifica (astrazione)	Dimostrata come catena di flusso dati (4 livelli)
Cosa vede l'LLM	Specifica di comportamento inferita (+ test falliti)	Referto causale + i soli 2 metodi rilevanti
Difesa dalla mappa vuota	Raffinamento della specifica sui test	Loop di validazione sui test
Patch finale	`return null` (corretta)	`return null` (corretta)
Supera i test?	Sì	Sì
È davvero corretto?	Sì	Sì

A differenza dell'esempio "da manuale" in cui l'IA fallisce, qui entrambi i sistemi *ben progettati* arrivano alla soluzione giusta, ed è proprio questo a renderlo istruttivo. Il vero spartiacque non è "IA contro ibrido" in termini di test superati, ma la **trappola della mappa vuota**: una patch che "sembra giusta" e in cui un approccio ingenuo o puramente code-centric cadrebbe facilmente. Ciò che protegge da quella trappola non è quanto è "intelligente" l'IA, ma la **qualità delle informazioni** e il rigore del processo: VibeRepair ricostruisce l'intento *inferendo* una specifica; HybridRepair lo *garantisce* con una diagnosi formale e autonoma e con una validazione che riesegue i test. La differenza, in definitiva, è nel *livello di garanzia*, inferenza contro dimostrazione, localizzazione fornita contro localizzazione autonoma, in modo riproducibile.

2.2 Strumenti fondamentali dell'approccio ibrido

Per superare i limiti evidenziati nei paragrafi precedenti e concretizzare l'idea di un approccio ibrido, l'architettura di *HybridRepair* poggia sull'integrazione di tre strumenti fondamentali. Ognuno di essi è stato selezionato per colmare le lacune degli altri, definendo una separazione dei compiti tra analisi formale, manipolazione strutturale e sintesi generativa.

2.2.1 LogicFL: Ragionamento Formale e Deduttivo

Il primo pilastro del sistema è *LogicFL*, un recente strumento di *Fault Localization* specificamente progettato per identificare le cause profonde delle *NullPointerException*. Utilizza la programmazione logica (Prolog) per imitare fedelmente il processo di deduzione umana impiegato dagli sviluppatori durante il *debugging* [22].

Combinando l'analisi statica della struttura del codice con l'analisi dinamica dei valori osservati durante l'esecuzione dei test falliti, LogicFL popola una *Knowledge Base* in SWI-Prolog su cui un motore di inferenza traccia a ritroso la catena causale che lega l'espressione del crash all'origine esatta del valore *null* [22]. Il dettaglio del meccanismo è approfondito nel Capitolo 3.

All'interno di *HybridRepair*, LogicFL funge da *oracolo dei guasti*: è un componente che, dato un test fallito, restituisce in modo deterministico un verdetto sulla causa del difetto, che le fasi successive assumono come dato di fatto e non come ipotesi da verificare. Fornisce così all'intelligenza artificiale un referto causale privo di allucinazioni. Questo, però, non lo rende *infallibile*: il referto è una catena di cause candidate ordinate per priorità euristica, e nulla garantisce che la causa reale venga sempre individuata o posta in cima. Il determinismo ne assicura la *riproducibilità*, non la correttezza assoluta. È proprio questa rigorosa tracciabilità causale a motivarne l'impiego come "guida" per i modelli generativi: l'adozione di *LogicFL* persegue, in particolare, un duplice e vitale scopo.

- **Fornire al motore generativo (LLM) un contesto semantico mirato:** estraendo l'esatta catena logica e di esecuzione (i "fatti"), *LogicFL* solleva l'LLM dal compito di navigare ciecamente nell'intero codice sorgente. Tracciando esplicitamente come il valore *null* viene trasferito all'interno del flusso di controllo, si segue a ritroso la sola catena che ha prodotto il crash, isolando esclusivamente le dipendenze realmente coinvolte. In questo modo si offre al modello un perimetro logico inequivocabile su cui operare, completamente depurato dall'eccesso di contesto che tipicamente causa disorientamento.
- **Restringere drasticamente lo spazio di ricerca e le possibilità di allucinazione:** l'applicazione delle regole deduttive dello strumento funge da filtro inflessibile contro le associazioni spurie. Identificando con precisione le sole espressioni che causano o trasferiscono il difetto, si guida attivamente l'LLM verso la sintesi di una *patch* sulle sole variabili formalmente rilevanti [22]. Ciò garantisce un intervento semanticamente e logicamente fondato, anziché limitarsi a un'ipotesi puramente statistica.

2.2.2 Tree-sitter: Parsing Incrementale ad Alta Fedeltà

Mentre LogicFL fornisce le coordinate logiche del difetto, la traduzione di queste direttive in un contesto comprensibile per il modello generativo e l'applicazione materiale della *patch* richiedono uno strumento di *parsing* del codice estremamente preciso. Per questo compito, l'architettura integra *Tree-sitter* [7], un generatore di parser incrementale ad altissime prestazioni. Come gli altri strumenti descritti in questa sezione, è una tecnologia esterna preesistente, integrata in *HybridRepair* ma non sviluppata in questo lavoro.

A differenza dei tradizionali analizzatori statici che producono un *Abstract Syntax Tree* scartando informazioni "superflue" come spaziature, formattazione e commenti, Tree-sitter genera un *Concrete Syntax Tree* (CST) [2]. Il CST mantiene una fedeltà assoluta al file testuale originale, registrando le coordinate spaziali esatte (riga e colonna) di ogni singolo byte e nodo sintattico. Questa caratteristica è di vitale importanza per *HybridRepair*: permette di estrarre chirurgicamente il frammento di codice rile-

vante (isolandolo dal resto del file) per fornirlo al *prompt* dell'LLM senza distorcere la visualizzazione originale. Soprattutto, garantisce che l'applicazione della *patch* finale generata dal modello avvenga tramite manipolazioni dirette dell'albero sintattico. Questo previene i classici errori di corruzione del codice (come parentesi sbilanciate o formattazione rotta) tipici degli strumenti che applicano le *patch* tramite semplici differenze testuali (*diff*) o espressioni regolari.

2.2.3 Large Language Models (GPT-4o) e Function Calling

L'ultimo pilastro è costituito dal motore generativo, istanziato mediante un *Large Language Model*, nello specifico GPT-4o, (lo stesso modello utilizzato da VibeRepair [57]). All'interno della pipeline, il modello è deliberatamente sollevato dal compito di localizzare il guasto, operazione per la quale risulta incline all'errore. Il suo ruolo è invece rilegato esclusivamente alla fase di *sintesi* e *ragionamento semantico* a valle della diagnosi formale.

Una caratteristica fondamentale che motiva l'uso di modelli come GPT-4o in questa architettura è il supporto nativo al paradigma del *Function Calling* (o *Structured Outputs*). Affinché un LLM possa agire in modo iterativo e autonomo all'interno di una Macchina a Stati Finiti (FSM) progettata per compilare e testare il codice, è imprescindibile che le sue risposte non siano semplice testo discorsivo, ma strutture dati rigorosamente tipizzate e interpretabili algoritmicamente (ad esempio, formati JSON validati). Grazie a questa funzionalità, il modello può generare le specifiche di riparazione (*RepairSpec*) e il codice correttivo seguendo i vincoli di interfaccia richiesti da *HybridRepair*, chiudendo così il cerchio tra la deduzione logica di Prolog, la manipolazione strutturale di Tree-sitter e la flessibilità generativa dell'Intelligenza Artificiale.

2.3 Defects4J: il benchmark di valutazione

L'ultimo elemento di contesto necessario prima di affrontare l'architettura riguarda i *dati* su cui HybridRepair viene sviluppato e messo alla prova. Un sistema di riparazione automatica ha senso solo se valutato su difetti *reali*, riproducibili e confrontabili con quelli usati dagli altri lavori della letteratura, per questo motivo si è scelto **Defects4J** [20].

2.3.1 Che cos'è Defects4J?

Defects4J è una raccolta curata di *bug* reali, estratti dalla storia di sviluppo di progetti *open source* Java molto noti (tra cui JFreeChart, Google Closure Compiler, Apache Commons Lang, Apache Commons Math e Jackson).

Per ciascun difetto, il *framework* mette a disposizione un ambiente di esecuzione riproducibile e, in particolare, quattro elementi fondamentali:

- la **versione difettosa** (*buggy*) del codice sorgente, ovvero il programma immediatamente prima della correzione;
- la **versione corretta** (*fixed*), che differisce dalla precedente per la sola *patch* applicata dagli sviluppatori e funge da soluzione di riferimento (*ground truth*);

- almeno un **test che innesca il difetto** (*triggering test*), cioè un caso di test che fallisce sulla versione *buggy* e passa su quella *fixed*;
- la **suite di test di regressione** associata, usata per verificare che la correzione non comprometta il comportamento preesistente.

Un aspetto centrale è l'**isolamento del difetto**: la differenza tra versione *buggy* e *fixed* è minimizzata e ricondotta al solo bug in esame, depurata da modifiche estranee (refactoring, cambi di formattazione, aggiornamenti non correlati). Questo consente di attribuire in modo inequivocabile un eventuale successo o fallimento alla *patch* generata, e non a fattori di contorno.

2.3.2 Identificazione dei difetti e sottoinsieme NPE

Ogni difetto è identificato da una sigla nella forma `Progetto-Numero` — ad esempio `Chart-2`, `Closure-2` o `Codec-13` — dove la prima parte indica il progetto di provenienza e la seconda l'indice del bug all'interno di quel progetto. È questa la notazione che il sistema riceve in ingresso (il `bug_id`) e che ricorre nei capitoli successivi quando si illustrano casi concreti di riparazione.

Poiché questa tesi si concentra sulle *NullPointerException* (per le ragioni discusse nel Capitolo precedente), non si utilizza l'intero benchmark, bensì il **sottoinsieme dei difetti riconducibili a NPE**: bug il cui *triggering test* fallisce proprio per via di una dereferenziazione di un riferimento `null`. Questo sottoinsieme costituisce il dominio di valutazione di HybridRepair.

2.3.3 Il ruolo di Defects4J in HybridRepair

All'interno del nostro lavoro Defects4J svolge un duplice ruolo, coerente con la *pipeline* descritta nei capitoli seguenti:

- come **sorgente di input**, fornisce per ogni `bug_id` il codice sorgente difettoso e i test falliti, ovvero esattamente il materiale da cui parte la Fase 1 (FaultOracle) per la localizzazione causale;
- come **oracolo di valutazione**, fornisce sia i test (che certificano se la *patch* risolve davvero il difetto senza introdurre regressioni) sia la versione *fixed* di riferimento, utile per un confronto qualitativo tra la correzione generata e quella umana.

Capitolo 3

HybridRepair

3.1 Obiettivi e requisiti del sistema

I capitoli precedenti hanno individuato il nodo critico del problema, l'incapacità degli approcci APR tradizionali e degli LLM privi di guida formale di affrontare la natura inter-procedurale delle *NullPointerException*, e hanno indicato nell'ibridazione neuro-simbolica la direzione di soluzione. Prima di descrivere come questa direzione si materializzi in un'architettura concreta, è necessario operazionalizzare l'obiettivo generale in obiettivi di dettaglio verificabili e nei requisiti che lo strumento realizzato deve soddisfare. Tali obiettivi e requisiti costituiscono il riferimento rispetto al quale, nel seguito del capitolo, verranno motivate le scelte architetturali e, nel Capitolo 4, valutati i risultati sperimentali.

3.1.1 Obiettivi di dettaglio

A partire dall'obiettivo generale, la riparazione automatica di difetti NPE reali guidata da una diagnosi formale, la ricerca persegue i seguenti obiettivi specifici:

O1 Riparazione *fully-automated* di NPE reali. Risolvere, senza alcun intervento umano nel ciclo, *bug* di tipo *NullPointerException* estratti dal *benchmark* standard Defects4J, a partire dal solo identificatore del difetto.

O2 Disaccoppiamento tra diagnosi e generazione. Separare nettamente l'inferenza *deduttiva* delle cause, e deterministica nel senso di *riproducibile*, così da contenere le allucinazioni logiche del modello e mantenere mirato e semanticamente fondato il suo intervento.

O3 Tracciabilità e spiegabilità della diagnosi. Garantire che il processo diagnostico sia sempre esplicabile, ancorando il ragionamento a regole logiche (Prolog) verificabili e ripetibili anziché a sole euristiche probabilistiche.

O4 Generazione vincolata a specifiche inequivocabili. Fornire all'LLM una specifica di riparazione priva di ambiguità e ancorata alle sole API realmente disponibili, riducendo lo spazio di ricerca e prevenendo la sintesi di codice plausibile ma logicamente scorretto.

3.1.2 Requisiti funzionali e non funzionali

Gli obiettivi precedenti si traducono in un insieme di requisiti che lo strumento deve soddisfare, distinti in funzionali (*ciò che il sistema deve fare*) e non funzionali (*le qualità con cui deve farlo*).

Requisiti funzionali.

RF1 Localizzare il *fault* nel codice sorgente Java e derivarne una diagnosi causale strutturata, basata su analisi statica e ragionamento deduttivo, senza ricorso a modelli generativi.

RF2 Trasformare la diagnosi formale in una specifica di riparazione in linguaggio naturale, ancorata al contesto API disponibile.

RF3 Generare, applicare, compilare e testare la *patch* operando su una *sandbox* isolata del sorgente.

RF4 Apprendere dai tentativi falliti tramite un meccanismo di *retry* adattivo, fino al raggiungimento di una *patch* valida o all'esaurimento del *budget*.

RF5 Produrre, a ogni fase, artefatti tracciabili e tipizzati che documentino l'esito e lo storico del processo.

Requisiti non funzionali.

RNF1 Automatismo. L'intera *pipeline*, dal `bug_id` alla *patch*, deve eseguire senza intervento umano.

RNF2 Determinismo della diagnosi. La fase di analisi causale deve essere ripetibile e indipendente dalla variabilità stocastica del modello generativo.

RNF3 Modularità e disaccoppiamento. Le fasi devono avere responsabilità coese e confini ben delineati, in modo da essere testabili in isolamento e sostituibili.

RNF4 Interfacce tipizzate. Le strutture dati che mediano le transizioni fra le fasi devono essere fortemente tipizzate e centralizzate, per resistere agli errori di disallineamento.

RNF5 Spiegabilità. La catena diagnostica deve restare ispezionabile e giustificabile in ogni suo passo.

RNF6 Riproducibilità. L'esecuzione su un dato difetto deve essere riproducibile, condizione necessaria alla validazione sperimentale.

3.2 Panoramica architetturale del sistema

3.2.1 Principi di progettazione

Coerentemente con la separazione tra diagnosi e sintesi argomentata nel Capitolo precedente e con gli obiettivi di disaccoppiamento, tracciabilità e generazione vincolata (O2–O4), l'architettura di HybridRepair traduce il principio neuro-simbolico in una struttura concreta: una componente simbolica (LogicFL su Prolog) presiede alla diagnosi deterministica del difetto, mentre una componente neurale (GPT-4 via Azure

OpenAI) ne traduce l'esito in codice corretto. A partire da un identificatore di difetto (ad esempio, Codec-13), il sistema esegue tre fasi sequenziali e nettamente disaccoppiate FaultOracle, SpecReason e PatchAgent, ciascuna con responsabilità coese, interfacce tipizzate e confini ben delineati.

3.2.2 *Pipeline* dei dati: input e output delle tre fasi

Per garantire che i moduli siano testabili in isolamento e sostituibili, il passaggio di informazioni tra una fase e la successiva è governato da un contratto di interfaccia esplicito. Le strutture dati che mediano queste transizioni sono fortemente tipizzate e centralizzate in un unico modello dati condiviso, realizzando un'architettura robusta e resistente agli errori di disallineamento.

La Figura 3.1 ne offre una visione d'insieme: le tre fasi sequenziali, gli artefatti tipizzati che le collegano e la natura simbolica o neurale di ciascuna.

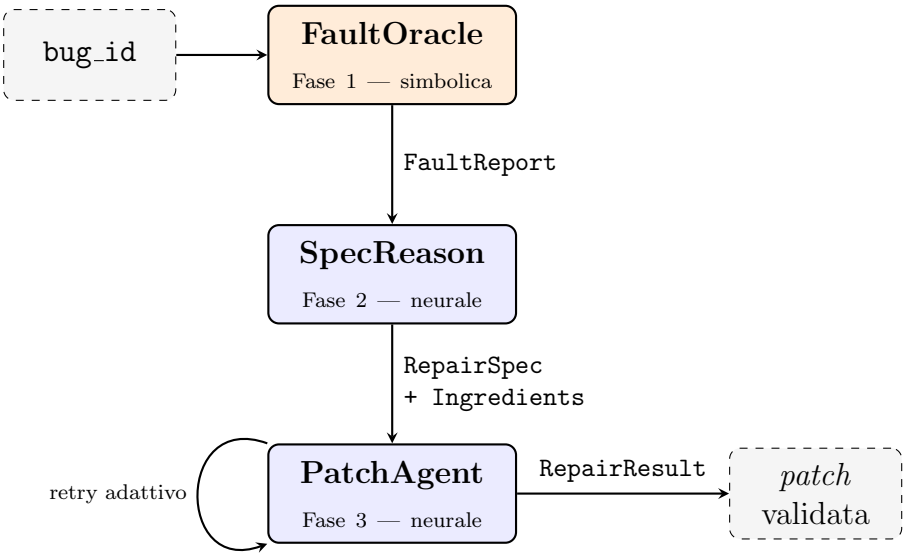

Figura 3.1: Visione d'insieme dell'architettura a tre fasi di HybridRepair. In arancio la fase simbolica deterministica (FaultOracle, basata su LogicFL/Prolog), in blu le fasi neurali generative (SpecReason, ragionamento *Chain-of-Thought*; PatchAgent, agente LLM a stati finiti con *retry* in *sandbox*); le etichette sugli archi indicano gli artefatti tipizzati scambiati tra una fase e la successiva.

La Tabella 3.1 riassume, per ciascuna delle tre fasi, le informazioni in ingresso, la natura dell'elaborazione e l'artefatto prodotto. I paragrafi successivi ne descrivono il ruolo concettuale; il funzionamento interno di ogni fase è trattato in dettaglio nelle sezioni dedicate (3.3, 3.4 e 3.5).

Tabella 3.1: Input, elaborazione e output delle tre fasi della *pipeline*.

Fase	Input	Elaborazione	Output
1. FaultOracle	`bug_id` e artefatti LogicFL pre-generati (`fault_locs`, `root_cause`, `stack_traces`)	Analisi statica e deduzione logica (Prolog); nessun modello generativo	`FaultReport`
2. SpecReason	`FaultReport`	Raccolta del contesto API + ragionamento *Chain-of-Thought* con auto-critica; nessun codice prodotto	`RepairSpec` + `Ingredients`
3. PatchAgent	`FaultReport`, `RepairSpec`, `Ingredients`	Agente LLM iterativo (FSM) che genera, applica, compila e testa la *patch* in *sandbox*	`RepairResult`

Fase 1 — FaultOracle. Ha l'onere esclusivo di localizzare il difetto nel codice sorgente Java e di derivarne una diagnosi causale strutturata. L'operazione esclude categoricamente i modelli generativi, affidandosi solo ad analisi statica e ragionamento deduttivo: l'analisi causale deterministica deve sempre precedere quella generativa, per evitare che l'LLM "inventi" problemi inesistenti. L'esito è un oggetto `FaultReport` che aggrega le locazioni del *fault* ordinate per sospiciosità, le catene causali (grezze e semanticamente ancorate) e il contesto dinamico del fallimento.

Fase 2 — SpecReason. Opera la transizione dal dominio formale della diagnosi a quello semantico-generativo: la diagnosi di Fase 1 viene trasformata in una specifica di riparazione in linguaggio naturale attraverso un ragionamento *Chain-of-Thought*. In questa fase *non si produce ancora codice*; l'obiettivo è la sintesi di una comprensione ragionata del problema, ancorata alle sole API realmente disponibili e validata da un passo di auto-critica. L'output è la coppia `RepairSpec` + `Ingredients`.

Fase 3 — PatchAgent. Materializza la comprensione in codice eseguibile. Un agente LLM, governato da una Macchina a Stati Finiti, genera, applica, compila e testa la *patch* operando su una *sandbox* isolata del sorgente, con un meccanismo di *retry* adattivo che apprende dai fallimenti precedenti. Il processo culmina nell'oggetto `RepairResult`, che documenta l'esito, il tentativo vincente e lo storico completo dei tentativi.

3.3 Fase 1 - FaultOracle e Localizzazione Causale

La prima fase di HybridRepair, denominata **FaultOracle**, risponde a una domanda precisa: *dove e perché* il programma difettoso produce un comportamento anomalo. Il termine "oracolo" è mutuato dalla teoria del software testing, dove indica un meccanismo capace di stabilire la correttezza di un output rispetto alle specifiche attese. FaultOracle opera qui come **oracolo diagnostico**: non si limita a rilevare il fallimento, ma emette una rappresentazione strutturata della radice causale del difetto, consumata dalle fasi successive della pipeline.

Il motore di inferenza sottostante è **LogicFL** [22], un sistema di *fault localization* basato su ragionamento logico-deduttivo, implementato in SWI-Prolog. A differenza dei localizzatori spettro-basati tradizionali (tecniche che stimano la sospettosità del codice dalla correlazione statistica tra copertura dei test ed esiti), LogicFL costruisce esplicitamente delle *catene causali*: tracce formali della propagazione di un valore anomalo (un riferimento `null`) dalla sorgente originaria fino al crash a runtime. Il processo deduttivo opera su una *knowledge base* di fatti Prolog, derivati dall'analisi statica della struttura del codice e dall'analisi dinamica dell'esecuzione dei test.

La Figura 3.2 offre una visione d'insieme del flusso dei dati: dagli input grezzi (codice sorgente Java e test JUnit fallenti) fino all'oggetto `FaultReport` strutturato e consumato dalle fasi successive della pipeline.

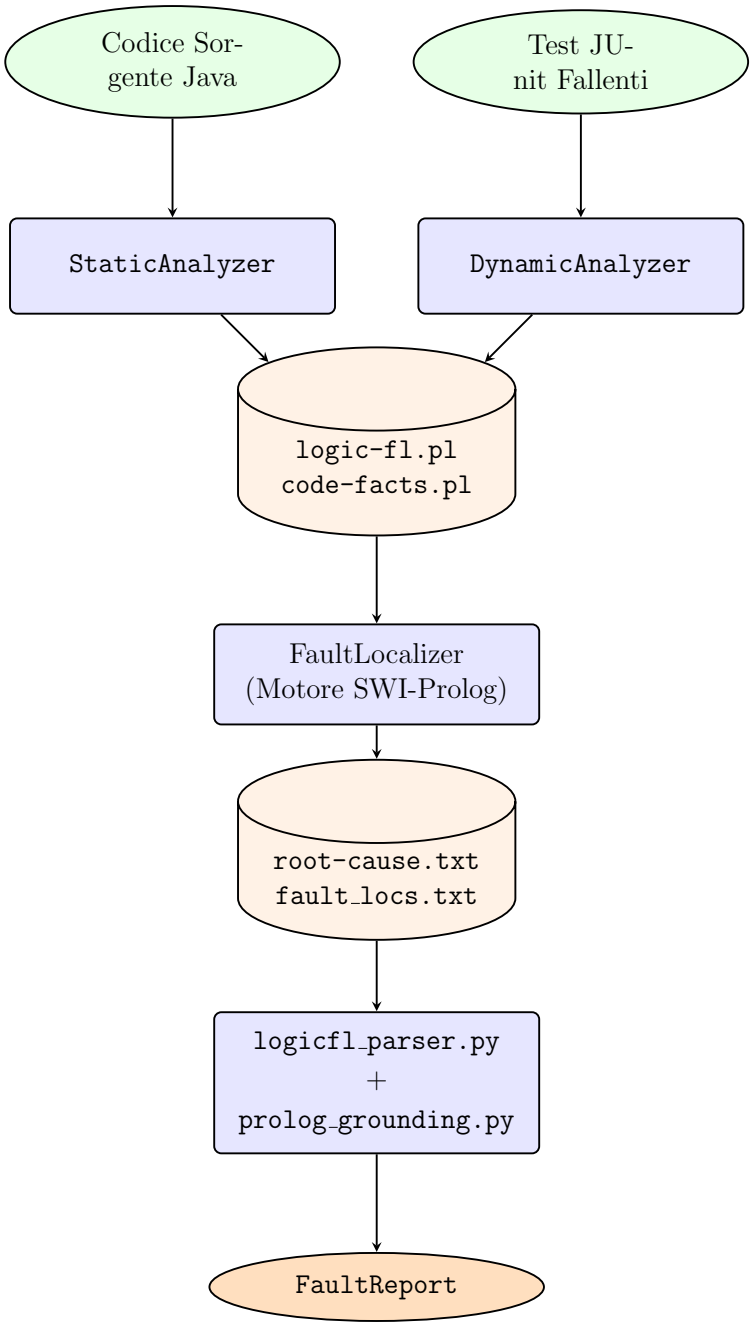

Figura 3.2: Flusso dei dati di FaultOracle: dall'input grezzo all'oggetto `FaultReport`. *Legenda*: ellissi verdi = dati in ingresso; rettangoli = componenti software; cilindri = artefatti su file; ellisse arancione = output strutturato.

Esempio Guida: Il Bug Csv-4
Per rendere concreti i concetti astratti di FaultOracle, l'intera trattazione seguente utilizzerà come *running example* il bug **Csv-4** dal dataset Defects4J. Questo difetto causa una `NullPointerException` nel progetto Apache Commons CSV quando si tenta di costruire una mappa a partire da header mai inizializzati. Le quattro righe che ci interessano sono:

```
// org.apache.commons.csv.CSVParser

// Nel costruttore: il campo viene riempito col risultato del
   metodo
this.headerMap = this.initializeHeader();     // ASSEGNAZIONE (
   Riga 244)

public Map<String, Integer> getHeaderMap() {
    // CRASH (Riga 288): this.headerMap e' null
    return new LinkedHashMap<String, Integer>(this.headerMap);
}

private Map<String, Integer> initializeHeader() throws
   IOException {
    Map<String, Integer> hdrMap = null; // ORIGINE (Riga 325)
    /* ... se non ci sono header definiti, hdrMap resta null
       ... */
    return hdrMap; // PROPAGAZIONE (Riga 351)
}
```

Prima di qualunque formalismo, proviamo a ragionare come farebbe uno sviluppatore, *all'indietro* a partire dal crash: il programma si interrompe a riga 288 perché `headerMap` è `null`; ma `headerMap` era stato riempito a riga 244 col risultato di `initializeHeader()`; quel metodo, a riga 351, restituisce `hdrMap`; e `hdrMap` era stato posto a `null` a riga 325. **Quella è l'origine.** Tutto ciò che LogicFL fa è automatizzare *esattamente* questa risalita: i fatti e le regole Prolog delle pagine seguenti non sono che la codifica formale di questo ragionamento.

3.3.1 Il Motore LogicFL: Analisi Statica e Dinamica

Architettura a tre componenti

L'architettura di LogicFL è modulare e si articola in tre componenti che operano in sequenza. Ciascun modulo è una classe Java del package `logicfl.analyzer.*`, invocabile tramite un corrispondente script shell.

Componente	Script	Classe Java	Output principale
StaticAnalyzer	`static_analyzer.sh`	`StaticAnalyzer`	`logic-fl.pl`, `code-facts.pl`
DynamicAnalyzer	`dynamic_analyzer.sh`	`DynamicAnalyzer`	Arricchimento con `val/3`
FaultLocalizer	`localizer.sh`	`FaultLocalizer`	`root-cause.txt`, `fault_locs.txt`

Il `FaultLocalizer` gira su SWI-Prolog, istanziato dentro la JVM tramite il bridge JPL (*Java–Prolog Library*).[1] Attraverso questo ponte la classe Java carica i file Prolog, vi asserisce dinamicamente nuovi fatti e valuta le query, raccogliendone le soluzioni in strutture dati native.

Analisi Statica: la Generazione dei Fatti Strutturali

Il primo componente, lo `StaticAnalyzer`, riceve il codice sorgente Java e ne costruisce una rappresentazione formale dell'Abstract Syntax Tree (AST), codificata come fatti Prolog. Il programma non viene eseguito: la sua struttura sintattica e relazionale è tradotta in clausole della logica del primo ordine (*code instrumentation* statica). Il risultato sono i file `logic-fl.pl` e `code-facts.pl`, base strutturale della *knowledge base*. Il vocabolario dei fatti copre gli elementi della semantica Java rilevanti per tracciare la propagazione dei valori `null`; la Tabella 3.2 ne riassume i blocchi fondamentali.

Tabella 3.2: I blocchi di fatti Prolog generati dall'analisi statica.

Predicato	**Semantica**
`assign/3`	Una variabile riceve il valore di un'espressione a una data riga.
`method_invoc/3`	Collega il risultato di una chiamata al metodo invocato; abilita il tracciamento inter-procedurale dei valori `null`.
`argument/3`	Argomento effettivo passato in posizione N a un'invocazione.
`param/3`	Parametro formale in posizione N nella firma del *callee*; con `argument/3` mappa argomenti effettivi e parametri formali tra scope diversi.
`return/3`	Espressione di ritorno di un metodo; usato per dedurre se un metodo può restituire `null`.
`ref/3`	Dereferenziazione di un nome in un'espressione: punto d'ingresso per il rilevamento delle *null dereference*.
`class/2`, `method/2`	Metadati: nome pienamente qualificato della classe e intervallo di righe del metodo (`range/5`), consultati dagli algoritmi di ranking causale.
`literal/6`, `name_ref/4`	Valori costanti noti a compile-time (`null`, interi, stringhe) e tipizzazione del nome (`var`, `param`, `field`).

Per dare concretezza al vocabolario della tabella, ecco come l'analisi statica codifica l'assegnazione e l'uso del campo `headerMap` in Csv-4:

```
assign(f_header_map_69, csvparser_1_expr20, line(csvparser_1,
    244)).
return(csvparser_1_expr23, m_get_header_map_61, line(
    csvparser_1, 288)).
argument(f_header_map_69, 1, csvparser_1_expr23).
```

Argomenti come `f_header_map_69` o `csvparser_1_expr20` sono identificatori interni opachi generati dall'analizzatore: un prefisso ne segnala il tipo (`f_` per i campi, `m_` per i metodi, il suffisso `_expr` per le espressioni) e un numero finale disambigua le occorrenze. Nel nostro caso `f_header_map_69` è il campo `headerMap`, `csvparser_1_expr20`

[1] Concretamente, la JVM carica la libreria nativa `libjpl.so` (direttiva `-Djava.library.path`), che apre un canale bidirezionale tra Java e SWI-Prolog.

è l'espressione `this.initializeHeader()` che lo inizializza e `m_get_header_map_61` è il metodo `getHeaderMap`; ricondurre questi simboli ai rispettivi nomi Java leggibili sarà compito del *Semantic Grounding* (Sezione 3.3.3). Ricorre inoltre, in quasi ogni fatto, il termine posizionale `line(ClassId, LineNum)`, che àncora l'evento a una precisa coppia (classe, riga): un riferimento univoco all'interno di un progetto multi-file, indispensabile per discriminare omonimie tra classi differenti.

Per il resto, questi fatti descrivono soltanto la *struttura* del codice: stabiliscono che `headerMap` viene assegnato a riga 244 e usato a riga 288, ma non dicono nulla sul fatto che a runtime valga davvero `null`. Quell'informazione arriva dall'analisi dinamica descritta di seguito; è solo la combinazione dei due tipi di fatto a fondare la deduzione del difetto.

Analisi Dinamica: la Generazione dei Fatti di Valore

L'analisi statica descrive la struttura del codice ma, per costruzione, non conosce i *valori concreti* che le variabili assumono durante l'esecuzione. Per asserire che un'espressione valuti effettivamente a `null` in un dato punto del flusso, durante il run di un test fallente, serve un'analisi dinamica.

Questo compito è demandato al `DynamicAnalyzer`, che lavora a livello di *bytecode*: il codice Java viene compilato normalmente, dopodiché la JVM lo esegue con un agente che inietta istruzioni di osservazione (senza modificare i file sorgente) nei punti in cui le variabili vengono lette o scritte. Ogni volta che il test fallente attraversa una di quelle istruzioni, il valore osservato viene registrato. L'esito è l'aggiunta, in coda a `logic-fl.pl`, di fatti dinamici della forma `val(Entità, Valore, Posizione)`: ciascuno attesta il valore concreto che un'entità ha assunto in un preciso punto dell'esecuzione. Per Csv-4 il fatto chiave è:

```
val(f_header_map_69, null, line(csvparser_1, 288)).
```

Generato a tempo di esecuzione durante il test fallente, questo predicato attesta che il campo `f_header_map_69` valeva `null` quando lo stack frame attraversava la riga 288. A differenza dei fatti strutturali, `val/3` è una misurazione empirica dello stato del programma. È proprio la combinazione di fatti strutturali (`assign/3`, `ref/3`, `method_invoc/3`) e fatti dinamici (`val/3`) a fondare la base del ragionatore Prolog: i primi delineano le relazioni possibili tra le entità, i secondi ancorano la deduzione agli eventi realmente osservati nella riproduzione del fallimento.

Il `DynamicAnalyzer` formalizza anche lo stack trace catturato al momento del crash:

```
test_failure(failure_1, 'org.apache.commons.csv.CSVParserTest',
    'testNoHeaderMap').
trace(trace_3, trace_2, m_get_header_map_61, line(csvparser_1,
   288), failure_1, target).
```

Il predicato `test_failure/3` identifica univocamente l'evento di fallimento, esplicitandone classe e metodo, e funge da ancoraggio radice cui fanno riferimento tutti gli eventi registrati per quel crash. Il predicato `trace/6` cattura il singolo frame dello stack, codificando l'ID dell'invocazione, la classe e il metodo chiamati, la riga sorgente, il contesto di fallimento e un tag binario che marca il frame come `target` (codice dell'applicativo) o `non_target` (framework di testing come JUnit, o librerie JRE). Questa

categorizzazione è determinante per identificare la linea candidata come vera origine dell'anomalia.

3.3.2 Costruzione e Ranking delle Catene Causali

Il nucleo inferenziale di LogicFL (il file `npe-rules.pl`) risponde a una domanda precisa: quale entità del programma è nulla in un dato punto, e da dove proviene tale nullità?. Il procedimento si articola in quattro passi: **(1)** localizzare il sito del crash, **(2)** individuare l'espressione nulla, **(3)** risalire la catena causale fino all'origine, **(4)** ordinare le cause per priorità diagnostica. Il punto d'ingresso è il predicato `find_npe_cause`, che orchestra i quattro passi e restituisce le cause già ordinate.

Dal crash all'origine: la risalita causale

I primi tre passi ricostruiscono il cammino del valore `null`, dal sito del crash fino alla sua origine. La Figura 3.3 ne mostra il risultato su Csv-4: una catena di quattro anelli, dall'origine (riga 325) al crash (riga 288).

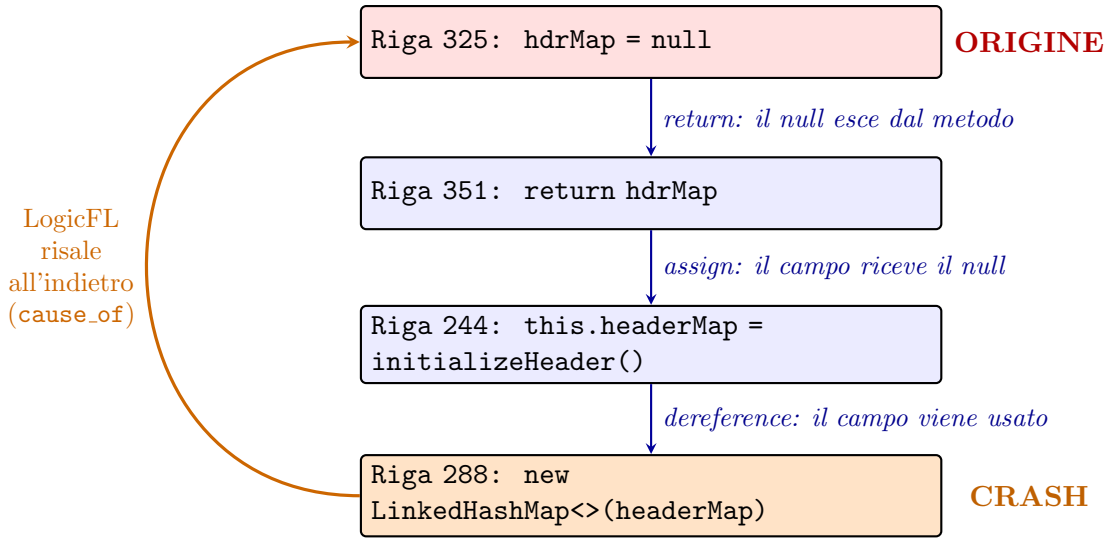

Figura 3.3: La catena causale di Csv-4. Il valore `null` nasce a riga 325 e si propaga in avanti fino al crash a riga 288; LogicFL lo scopre percorrendo la catena *all'indietro*, dal crash all'origine. Ogni arco corrisponde a una regola di trasferimento.

(1) Sito del crash e **(2) espressione nulla**. La riga del crash è il primo frame dello stack trace che appartiene al codice applicativo non a librerie o al framework di test; se il frame più interno è di libreria, il sistema risale al chiamante (essenziale per le eccezioni lanciate dentro metodi di libreria). Individuata la riga, LogicFL cerca l'espressione effettivamente nulla. In Csv-4 si tratta di una *dereferenziazione*: il campo `headerMap`, attestato come `null` dai fatti dinamici, viene usato a riga 288 (questo lo classificherà più avanti come `UNINITIALIZED_FIELD`). LogicFL gestisce anche un secondo caso, non presente in Csv-4, in cui il `null` non viene dereferenziato direttamente ma *passato come argomento* a un metodo.

(3) Risalita causale. È il cuore deduttivo di LogicFL, Il predicato `cause_of` prova in ordine tre ipotesi sempre più generali su *dove* sia la causa:

```
% Caso 1 - causa immediata al sito del crash
cause_of(npe(Expr, Line), Expr, Line) :- check_val(Expr, null,
   Line).

% Caso 2 - origine diretta (il null nasce qui)
cause_of(npe(Expr, Line), Cause, Loc) :- originated_from(Expr,
   Cause, Loc).

% Caso 3 - propagazione transitiva (segui la traccia all'
   indietro)
cause_of(npe(Expr, Line), Cause, Loc) :- can_be_transferred(
   Expr, Cause, Loc).
```

Il primo caso copre il difetto puramente locale (l'entità è già nulla alla riga del crash); il secondo individua il punto in cui il `null` nasce per la prima volta, in Csv-4 la riga 325; il terzo, quello interessante, segue la propagazione all'indietro di un passo, riconoscendo i tre modi in cui un `null` può trasferirsi tra espressioni: per *assegnazione*, per passaggio *parametro* ← *argomento* tra chiamante e chiamato, o tramite *valore di ritorno*. Sono esattamente gli archi della Figura 3.3: in Csv-4 il valore di ritorno collega il `return hdrMap` di riga 351 al suo chiamante, e l'assegnazione collega il campo a riga 244. Una ricerca in profondità, che tiene traccia dei nodi già visitati per non entrare in cicli, compone questi singoli passi in catene di lunghezza arbitraria.

Ranking e output

L'ultimo passo ordina le cause per rilevanza diagnostica in tre fasi. Il **Filtering** scarta le cause formalmente corrette ma inutilizzabili per costruire una patch: le origini interne al codice di test e i metodi puramente *delegati* (getter o wrapper che si limitano a restituire un parametro ricevuto), i quali segnalano che l'anomalia risiede più a monte nella catena di invocazioni. La **Preference** eleva in cima le cause più adatte alla generazione di una patch, cioè quelle che soddisfano almeno uno di tre pattern: un metodo con percorsi di ritorno sia `null` sia validi (*missing null-return guard*); un'assegnazione di `null` avvenuta localmente *prima* della riga del crash; oppure un metodo che è l'unico del codice applicativo presente nello stack trace. Infine la **Composizione** antepone le cause preferite alle rimanenti.

L'output è serializzato in due file. Il primo, `root-cause.txt`, elenca le cause individuate *in ordine di ranking* (non come cammino lineare): la prima voce coincide con il sito del crash, le successive sono le restanti cause in ordine di priorità.

```
NPE at line(csvparser_1, 288) / Null Expression -
   f_header_map_69[headerMap]
         can be caused by
f_header_map_69[headerMap] - line(csvparser_1, 288).
v_hdr_map_70[hdrMap] - line(csvparser_1, 351).
csvparser_1_literal5[null] - line(csvparser_1, 325).
csvparser_1_expr20['this.initializeHeader()'] - line(
   csvparser_1, 244).
```

Per Csv-4, la riga 351 (`return hdrMap`) compare subito dopo il sito del crash proprio perché la Preference la eleva, riconoscendo il pattern *missing null-return guard* in `initializeHeader()`. In ogni voce, il campo alfanumerico è l'identificatore interno

opaco del motore, mentre il testo tra parentesi quadre è il referente Java leggibile estratto dall'AST. Questa sequenza ordinata è preservata nel campo `causal_chains` del `FaultReport`, così che l'LLM possa esaminare le cause a partire dalla più probabile.

Il secondo file, `fault_locs.txt`, fornisce la vista compatta: le sole righe candidate, nello stesso ordine di priorità.

```
org.apache.commons.csv.CSVParser 288
org.apache.commons.csv.CSVParser 351
org.apache.commons.csv.CSVParser 325
org.apache.commons.csv.CSVParser 244
```

.

3.3.3 Semantic Grounding: Dalla Logica Formale all'Identità Java

Il Problema della Vaghezza Simbolica

L'output di LogicFL, per quanto rigoroso dal punto di vista logico, si ferma agli identificatori Prolog opachi (come `f_header_map_69` o `csvparser_1_expr20`): questi hanno pieno significato nel motore di inferenza ma sono privi di significato per un Large Language Model. Questa "vaghezza simbolica" fornisce al modello informazioni formalmente corrette ma semanticamente povere, aumentando il rischio di allucinazioni. Per risolverla è stato progettato e implementato, come contributo originale di questa tesi, il modulo `prolog_grounding.py`, che non fa parte di LogicFL e applica il *grounding semantico*: ogni simbolo Prolog viene tradotto nella sua controparte Java concreta (nome, tipo e ruolo strutturale), fornendo al modello un quadro diagnostico preciso, tipo-corretto e adatto alla generazione di una patch.

La Tassonomia degli Identificatori Prolog

Il primo passo del Semantic Grounding consiste nell'ispezionare l'identificatore Prolog per classificare il tipo di entità tramite specifiche convenzioni di denominazione interne. I prefissi come `v_`, `f_` o `param` identificano inequivocabilmente le variabili locali, i campi di classe o i parametri formali dei metodi. Questa classificazione rigorosa determina la corretta strategia di risoluzione AST successiva. Di seguito il blocco Python `_classify_entity_kind` che implementa questa logica essenziale:

```
def _classify_entity_kind(prolog_id: str) -> str:
    # Classificazione basata sull'ispezione del prefisso dell'
       identificatore Prolog
    if prolog_id.startswith("v_"):
        return "variable"
    elif prolog_id.startswith("param") or prolog_id.startswith(
       "p_"):
        return "parameter"
    elif prolog_id.startswith("f_"):
        return "field"
    elif "expr" in prolog_id or "literal" in prolog_id:
        return "expression"
    return "unknown"
```

La Risoluzione dei Tipi Tramite Tree-sitter AST

Per recuperare il tipo Java di ciascuna entità, il sistema usa la libreria Tree-sitter come parser incrementale, interrogando direttamente l'AST senza dover compilare l'intero progetto. Una funzione di smistamento (`_resolve_type_at_line`) instrada la richiesta al risolutore specializzato in base alla categoria calcolata in precedenza: variabili locali, parametri, campi o espressioni. I quattro risolutori condividono l'idea di fondo ma differiscono nello scope di ricerca; ne vediamo il caso rappresentativo, quello delle variabili locali.

Per le **entità dichiarate**, variabili locali, parametri formali e campi, la strategia è la stessa: raggiungere il nodo di dichiarazione nell'AST e leggerne il tipo. I tre casi differiscono solo nello *scope* di ricerca:

- le **variabili locali** si cercano nel solo metodo che contiene il crash, con una DFS che individua il nodo; poiché l'AST separa il tipo dal dichiaratore, si raccoglie il nodo del tipo (nelle sue varie forme: `type_identifier`, `generic_type`, `array_type`...) e il `variable_declarator` il cui nome coincide con la variabile cercata;
- i **parametri formali** sono più immediati: l'AST espone nome e tipo come campi (`name` e `type`) del nodo `formal_parameter`, accessibili con `child_by_field_name`, senza bisogno della DFS;
- i **campi di istanza/classe** usano la stessa logica delle variabili locali, ma estesa all'intero albero anziché al singolo metodo, perché le dichiarazioni di campo risiedono a livello di classe.

Per le **espressioni composte** (`entity_kind == "expression"`), la risoluzione esatta del tipo restituito non è possibile tramite una semplice ricerca di dichiarazione, poiché dipende dalla semantica dei metodi invocati. Pertanto, il sistema applica un'euristica mirata tramite pattern matching per riconoscere le operazioni di *cast esplicito*. L'espressione regolare è stata studiata per intercettare non solo tipi semplici, ma anche strutture sintatticamente più ricche, includendo costrutti Java avanzati (come tipi generici o array).

```
def _infer_type_from_expression(expr: str) -> str:
    """Inferisce il tipo di un'espressione tramite pattern
       matching."""
    # Pattern: (Tipo)espressione -> il tipo dichiarato nel cast
        è assunto come tipo
    cast_match = re.match(r"\((\w[\w<>\[\],\s]*)\)", expr)
    if cast_match:
        return cast_match.group(1).strip()
    return ""
```

Ad esempio, in un altro bug del dataset, data l'espressione `(XYItemRenderer)this.renderers.get(i)`, il pattern isola ed estrae la stringa `XYItemRenderer`.

Meccanismo di fallback testuale: qualora l'analisi tramite Tree-sitter non fosse disponibile o fallisse la ricerca nell'albero, il sistema possiede un meccanismo di fallback. Questa procedura compie una scansione testuale all'indietro (*backward search*) limitata alle 50 righe antecedenti la riga del crash, una finestra empiricamente sufficiente a

coprire la lunghezza tipica dei metodi analizzati nel benchmark Defects4J. L'espressione regolare utilizzata per questo compito supporta identificatori complessi, tenendo conto dell'eventuale modificatore `final`, tipi parametrici, array e path di package.

```
def _resolve_type_regex(lines, line_no, java_name, entity_kind)
    -> str:
    """Fallback regex per la risoluzione del tipo in backward
       search."""
    # Cattura tipi complessi ignorando l'eventuale modificatore
        'final'
    var_re = re.compile(
        rf"(?:final\s+)?([\w<>\[\],.?]+)\s+{re.escape(java_name
           )}\b"
    )
    search_start = max(0, line_no - 50)
    for i in range(line_no - 1, search_start - 1, -1):
        m = var_re.search(lines[i])
        if m:
            candidate = m.group(1).strip()
            # Successivamente il sistema esclude parole chiave
               riservate
            # (es. 'return') che potrebbero attivare falsi
               positivi
            return candidate
    return ""
```

L'approccio testuale è relegato a *fallback* perché, a differenza dell'analisi strutturale AST, le regex presentano limiti intrinseci:

- **Falsi positivi:** possono intercettare codice commentato, letterali stringa o parole chiave che mimano sintatticamente una dichiarazione;
- **Mancanza di scoping:** la scansione all'indietro è "cieca" rispetto all'annidamento dei blocchi e ai casi di *shadowing*, mentre l'AST mantiene piena consapevolezza dello scope;
- **Fragilità sintattica:** ritorni a capo arbitrari, generici annidati, array e annotazioni rendono le espressioni regolari complesse e fragili.

La procedura a ritroso funge quindi da rete di sicurezza per i fallimenti puntuali dell'analisi strutturale.

La Classificazione delle NPE

Prima della classificazione, un parser a espressioni regolari (in `prolog_grounding.py`) disassembla il formato testuale di `root_cause.txt`, già illustrato in precedenza, in record grezzi, uno per ciascun anello della catena causale, aggregati per sito di crash. È su questi record che opera la risoluzione dei tipi descritta sopra, producendo le entità Java tipizzate.

A partire dall'entità nulla e dalla lista delle cause, il sistema classifica poi ogni NPE in una di sei categorie. Le regole Prolog di `npe-rules.pl` (colonna centrale della tabella) appartengono a LogicFL; la tassonomia a sei categorie che le raggruppa in etichette

semantiche (colonna sinistra) è invece una mappatura originale di questa tesi, introdotta in `prolog_grounding.py` per rendere le cause interpretabili dall'LLM, e la sua completezza non è discussa né validata nel lavoro originale su LogicFL: nel paper di riferimento `null_ref/2` è un predicato unico, mai suddiviso in `field`/`indicizzazione`/ dereferenziazione generica come nella tabella seguente; `is_null_return/2` vi compare solo come condizione di preferenza nel ranking dei candidati, non come categoria di classificazione; e una categoria analoga a `NULL_LITERAL_ASSIGN` non è definita. Il sistema ispeziona le regole Prolog che hanno generato le cause e assegna la categoria corrispondente. La mappatura è riassunta nella tabella seguente (per Csv-4 la categoria risultante è `UNINITIALIZED_FIELD`):

Categoria Python	**Regola Prolog**	**Semantica**
NULL_RETURN	`is_null_return`	Il metodo ha percorsi di ritorno null e non-null
NULL_ARGUMENT	`null_arg_passed`	Un argomento null viene passato a un metodo
UNINITIALIZED_FIELD	`null_ref` (tipo field)	Un campo non è stato correttamente inizializzato
NULL_ARRAY_ACCESS	`null_ref` (su indicizzazione)	Un elemento specifico di array/collection è null
NULL_DEREFERENCE	`null_ref` generico	Dereferenziazione diretta e immediata di variabile nulla
NULL_LITERAL_ASSIGN	`val` con literal in cause	Assegnazione esplicita e diretta al valore null

La Struttura Dati GroundedNPE

Il risultato del Semantic Grounding è l'oggetto `GroundedNPE`, che struttura gerarchicamente gli strati informativi dell'inferenza. Questa gerarchia risiede in `shared/models.py` e usa le `dataclass` di Python per contratti tipizzati:

```
from dataclasses import dataclass
from typing import List

@dataclass
class GroundedEntity:
    """Un'entità Prolog risolta nella sua identità Java reale."""
    prolog_id: str      # Es. "f_header_map_69" (ID opaco Prolog)
    java_name: str      # Es. "headerMap" (nome Java leggibile)
    java_type: str      # Es. "Map<String,Integer>" (tipo risolto via AST)
    entity_kind: str    # Es. "field" (ruolo semantico)
    source_line: int    # Es. 288 (riga sorgente, 1-indexed)

@dataclass
class GroundedCause:
    """Una singola causa nella catena causale, con contesto
       Java risolto."""
    entity: GroundedEntity
```

```
    explanation: str     # Spiegazione testuale, es. "Explicitly
        assigned `null`"

@dataclass
class GroundedNPE:
    """Diagnosi NPE completamente grounded con classificazione
       e direttiva."""
    crash_line: int                # Riga in cui si verifica il
       crash
    class_id: str                  # ID identificativo della
       classe
    null_entity: GroundedEntity    # L'entità nulla nel sito del
        crash
    causes: List[GroundedCause]    # Catena ordinata delle cause
        radice
    category: NPECategory          # Classificazione tassonomica
        del bug
    repair_directive: str          # Direttiva generata per la
       riparazione
```

Questa gerarchia confluisce nel campo `grounded_chains` (una `List[GroundedNPE]`) dell'oggetto `FaultReport`, prodotto finale della Fase 1 e payload di input per la generazione dei prompt LLM.

Direttive di Riparazione (Rule-to-Prompt)

Sulla base della classificazione, il sistema genera una direttiva di riparazione specifica per categoria. Il dizionario `_REPAIR_DIRECTIVES` implementa il meccanismo "Rule-to-Prompt" tramite template parametrici:

```
_REPAIR_DIRECTIVES: Dict[NPECategory, str] = {
    NPECategory.NULL_RETURN: (
        "The method {cause_expr} can return null. "
        "Add a null-check on its return value before
           dereferencing, "
        "OR modify the method to guarantee a non-null return
           value "
        "(e.g., return an empty collection or a default object)
           ."
    ),
    NPECategory.NULL_ARGUMENT: (
        "The parameter `{null_name}` can be null when passed to
            this method. "
        "Add an early null guard at the top of the method "
        "(e.g., `if ({null_name} == null) return ...;`), "
        "OR ensure the caller never passes null."
    ),
    NPECategory.UNINITIALIZED_FIELD: (
        "The field `{null_name}` is not initialized (or
           explicitly set to null). "
        "Add initialization in the constructor or a lazy-init
           null-check before use. "
```

```
        "Consider what the correct default value should be."
    ),
    NPECategory.NULL_ARRAY_ACCESS: (
        "The element retrieved from `{null_name}` is null. "
        "Add a null-check on the retrieved element before
           dereferencing it, "
        "OR ensure the array/collection never stores null at
           that position."
    ),
    NPECategory.NULL_DEREFERENCE: (
        "The variable `{null_name}` is null when dereferenced
           at line {crash_line}. "
        "Add a null guard before the dereference, "
        "OR trace back why it is never assigned a non-null
           value."
    ),
    NPECategory.NULL_LITERAL_ASSIGN: (
        "The variable `{null_name}` is explicitly assigned `
           null` at line {assign_line} "
        "and later dereferenced at line {crash_line}. "
        "Change the null assignment to a safe default, or add a
            null-check "
        "before the dereference."
    )
}
```

Le direttive sono parametriche rispetto alle informazioni grounded: segnaposto come {`null_name`} vengono sostituiti dinamicamente con il nome Java reale dell'entità nulla. La parametrizzazione trasforma così un'analisi statica astratta in un'istruzione imperativa, direttamente comprensibile dal modello generativo. La direttiva così istanziata viene salvata nel campo `repair_directive` del rispettivo `GroundedNPE` (e quindi in `FaultReport.grounded_chains`): non è ancora il prompt completo, ma il frammento operativo che la fase di riparazione inserirà nel prompt inviato all'LLM.

Perché Questo Aiuta l'LLM

Il confronto tra l'output grezzo di LogicFL e quello arricchito dal grounding illustra il vantaggio di questa architettura.

Prima (Grezzo - Output di LogicFL)

```
NPE at line(csvparser_1, 288) / Null Expression -
   f_header_map_69[headerMap]
```

Dopo (Strutturato - Input per l'LLM)

```
Categoria: UNINITIALIZED_FIELD
Entita: headerMap (Tipo: Map<String, Integer>) alla riga 288
Direttiva: "The field `headerMap` is not initialized (or
   explicitly set
to null). Add initialization in the constructor or a lazy-init
null-check before use. Consider what the correct default value
   should be."
```

Ricevendo questo input mirato, l'agente LLM di HybridRepair è in grado di produrre, al suo secondo tentativo, una patch che supera la suite di test di Csv-4, introducendo un ternary operator che gestisce esplicitamente il caso non inizializzato:

```
// Patch generata da HybridRepair per Csv-4
public Map<String, Integer> getHeaderMap() {
    return this.headerMap == null ? null : new LinkedHashMap<>(
       this.headerMap);
}
```

La differenza qualitativa si articola su quattro dimensioni:

- **Risoluzione dei tipi:** l'output grounded include il tipo Java (`Map<String, Integer>`), assente nel testo grezzo; il modello può ragionare sul costruttore di classe più appropriato.
- **Classificazione semantica:** la categoria `UNINITIALIZED_FIELD` focalizza l'attenzione sul ciclo di vita del campo all'interno della classe, suggerendo una soluzione mirata.
- **Direttiva operativa:** la direttiva traduce l'analisi logica in un'istruzione d'azione concreta, guidando l'LLM ad implementare un null-check piuttosto che modifiche arbitrarie, garantendo una *constrained generation*.
- **Eliminazione dell'ambiguità simbolica:** gli identificatori opachi (`f_header_map_69`) sono sostituiti dalle controparti Java, evitando che il modello li confonda con nomi di variabili reali.

In sintesi, mentre i modelli puramente statistici operano *a posteriori* sul testo sorgente, HybridRepair combina la certezza deduttiva di Prolog con la flessibilità generativa del modello: il Semantic Grounding è il ponte tra logica e probabilità.

Robustezza e Degradazione Controllata

Il modulo `prolog_grounding.py` segue una filosofia di *graceful degradation*: ogni fase può fallire senza compromettere il sistema. Se Tree-sitter è irreperibile, si ricade sul fallback a regex; se il tipo resta irrisolto, il campo è omesso e la riparazione procede con una direttiva parzialmente istanziata; se `root_cause.txt` non è reperibile, si usano le catene causali grezze. Questo evita *single points of failure*.

3.3.4 Integrazione nella Pipeline Python: Il Modulo FaultOracle

L'integrazione del processo diagnostico nella pipeline Python è orchestrata dal modulo `fault_oracle/`, che media tra il motore Prolog e le fasi di riparazione neurale. Il modulo `logicfl_runner.py` gestisce il ciclo di vita dell'analisi LogicFL, riusando i risultati già disponibili per evitare riesecuzioni ridondanti. Terminato il motore Prolog, `logicfl_parser.py` legge i file generati e ne coordina l'arricchimento semantico (*Semantic Grounding*), consolidando localizzazioni, catene causali grezze e analisi AST nel report finale:

```
def parse_logicfl_output(bug_dir: Path) -> FaultReport:
    # 1. Estrazione delle localizzazioni con contesto AST
```

```
    locations = _parse_fault_locations(result_dir / "fault_locs
        .txt", source_roots)

    # 2. Parsing delle catene causali grezze prodotte da Prolog
    root_cause_text, causal_chains = _parse_root_cause(
        result_dir / "root_cause.txt")

    # 3. Risoluzione e arricchimento tramite Semantic Grounding
    grounded_chains = ground_causal_chains(
        root_cause_text, source_file=locations[0].file_path,
            ...
    )

    # Generazione e ritorno del report diagnostico unificato
    return FaultReport(
        bug_id=bug_id,
        locations=locations,
        root_cause_text=root_cause_text,
        causal_chains=causal_chains,
        grounded_chains=grounded_chains
    )
```

Il `FaultReport` finale è l'interfaccia strutturata tra la diagnostica e le fasi successive della pipeline: mettendo a disposizione righe sospette, spiegazione dell'errore e analisi causale, garantisce che la generazione della patch disponga di tutto il contesto necessario.

3.4 Fase 2 - SpecReason: dalla diagnosi alla specifica di riparazione

La seconda fase di `HybridRepair`, denominata **SpecReason**, costituisce il nucleo cognitivo del sistema: le informazioni diagnostiche prodotte dal `FaultOracle` vengono trasformate in conoscenza operativa, ossia una specifica di riparazione strutturata che guiderà il modello `GPT-4o` nella Fase 3. Il nome riflette la duplice vocazione: *Spec* per la produzione di una specifica rigorosa (la `RepairSpec`), *Reason* per il paradigma di ragionamento forzato con cui tale specifica è elicitata dal modello.

SpecReason mira a risolvere due famiglie di fallimento tipiche dell'APR basato su LLM, già discusse in precedenza:

- **L'allucinazione:** il modello genera codice sintatticamente plausibile ma invoca metodi o campi inesistenti nella libreria target.
- **La deriva semantica:** il modello interpreta il bug come un pattern di correzione familiare appreso nel pre-training, anziché come problema specifico dell'istanza, allontanandosi dalla logica richiesta.

Per mitigare queste criticità, SpecReason si avvale di due meccanismi complementari: l'**Ingredient Forge**, che raccoglie proattivamente il vocabolario reale delle API dal codice sorgente e lo inietta nel prompt, e il paradigma **Chain-of-Thought** (`spec_builder`

`.py`), che costringe il modello a esplicitare un ragionamento sul difetto prima di scriverne la correzione. Un terzo livello di salvaguardia, **SpecCritic** (`spec_critic.py`), implementa un ciclo di auto-critica in cui una seconda chiamata al modello valuta la coerenza logica della specifica confrontandola con le prove empiriche (stack trace, test falliti). L'artefatto finale è una `RepairSpec` validata, contesto per la Fase 3 (`PatchAgent`).

3.4.1 Il Flusso Dati

Il flusso di dati della Fase 2 è orchestrato dal modulo `repair_bug.py`, che coordina l'esecuzione sequenziale dei sotto-componenti:

```
# Frammento da repair_bug.py - orchestrazione della Fase 2

# Ingredient Forge
ingredients = collect_ingredients(fault_report)
# SpecBuilder (CoT)
spec = build_spec(fault_report, ingredients, client=client)
# SpecCritic
spec = validate_spec(spec, fault_report, client=client)
```

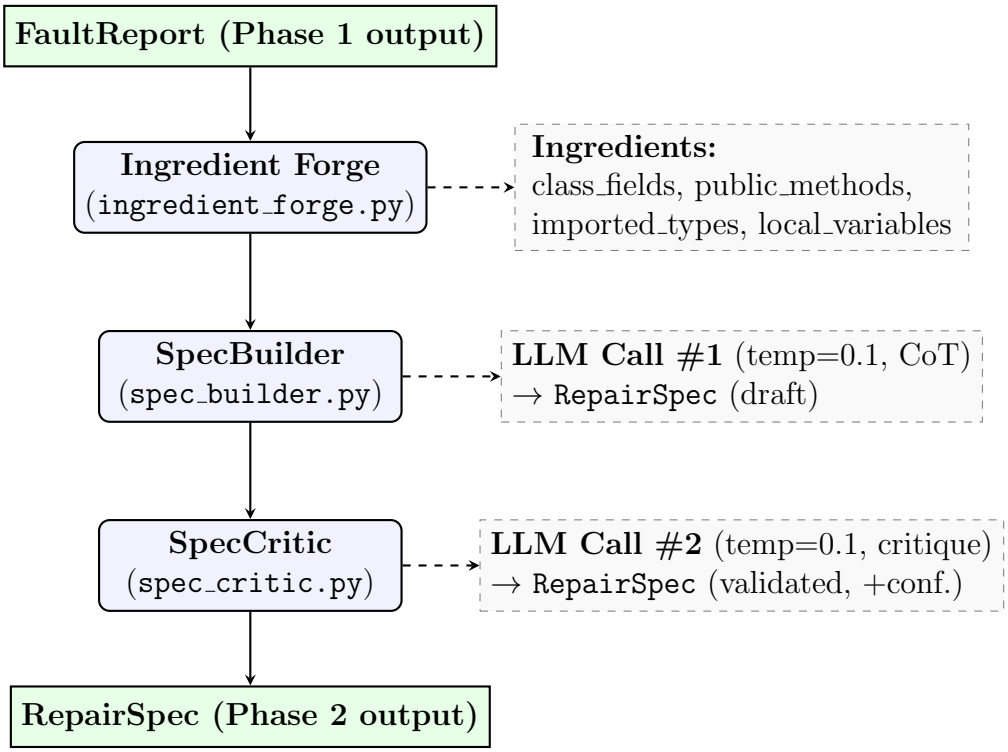

Figura 3.4: Architettura e flusso dei dati della Fase 2.

La Fase 2 costruisce il prompt di riparazione fondendo due fonti di informazione complementari. La prima è il `FaultReport` prodotto dalla Fase 1, che porta con sé le catene causali semanticamente ancorate di `LogicFL` (risolte da `prolog_grounding.py`): indicano al modello *dove* e *perché* si manifesta il difetto. La seconda sono gli "ingredienti" statici, i nomi di metodi, campi e firme realmente presenti nella classe, estratti dal codice sorgente da `ingredient_forge.py`: indicano al modello *con quali strumenti reali* costruire la patch. Combinando le due, ogni prompt resta radicato nella specificità del bug corrente, invece di affidarsi alla memoria parametrica del modello, che potrebbe suggerire pattern obsoleti o API inesistenti.

3.4.2 Ingredient Forge: la Raccolta Proattiva del Contesto Statico

L'allucinazione, la generazione di patch che invocano metodi o campi inesistenti nella libreria target, scaturisce dal fatto che il modello, addestrato su milioni di frammenti Java eterogenei, apprende le distribuzioni statistiche dei pattern di codifica ma non ha accesso all'interfaccia pubblica effettiva della classe da riparare. Per neutralizzarla, il modulo `ingredient_forge.py` implementa la *context injection*: anziché confidare nella memoria parametrica del modello, il vocabolario reale della classe viene estratto dal codice sorgente difettoso e iniettato nel prompt come vincolo esplicito. L'approccio trae ispirazione da sistemi come `FitRepair` [45] e `ReinFix` [51], che hanno mostrato come la pre-iniezione di codice reale riduca le allucinazioni; `HybridRepair` ne sistematizza l'idea estraendo molteplici categorie di ingredienti statici.

Architettura del Modulo ingredient_forge.py

`Ingredient_forge.py` si basa sulla funzione coordinatrice `collect()`, che orchestra una serie di funzioni private di analisi sintattica per produrre un oggetto `Ingredients`. La scelta metodologica più rilevante è l'uso di espressioni regolari (regex) per il parsing del sorgente grezzo, scartando i parser AST completi (Tree-sitter, JavaParser). La decisione è motivata da esigenze di robustezza: nell'APR il codice in input è per definizione difettoso e un parser AST rigido fallirebbe in modo irrecuperabile su input malformato, bloccando la pipeline. Le regex, invece, sono elastiche: ignorano il rumore di fondo ed estraggono firme di metodi e variabili anche con sintassi compromessa. Sono compilate staticamente una sola volta per processo.

```
# Estratto da ingredient_forge.py - compilazione statica dei
    pattern regex
_PUBLIC_METHOD_RE = re.compile(
    r"^\s*"
    r"(?:@\w+(?:\([^)]*\))?\s*)*"                          #
        optional annotations
    r"(public|protected)\s+"
    r"(?:static\s+|final\s+|abstract\s+|synchronized\s+|native\
        s+|strictfp\s+)*"
    r"(?:<[^>]+>\s+)?"                                     # generic
         params
    r"([\w<>\[\],.\\s?]+?)\s+"                              # return
         type
    r"(\w+)\s*\(([^)]*)\)",                                # name(
        params)
    re.MULTILINE,
```

Prima di tentare qualsiasi estrazione, il modulo applica però una fondamentale fase di pre-processing tramite la funzione `_strip_comments()`. Questa funzione rimuove sia i commenti in blocco (`/* ... */`) sia i commenti in linea (`// ...`) esplorando il testo con due passaggi separati. Eliminare i commenti è un passo critico per prevenire i falsi positivi: senza questa operazione preliminare, le regex estrarrebbero erroneamente firme di metodi menzionate a scopo documentale all'interno del `JavaDoc` o frammenti di codice "commentato" dagli sviluppatori.

Come mostra il listing, `_PUBLIC_METHOD_RE` non si limita a cercare nomi e parentesi: i gruppi opzionali assorbono annotazioni (`@Override`, `@Nullable`), modificatori (`static`, `final`, `synchronized`) e parametri di tipo generico a livello di metodo. Catturare l'intera firma è essenziale nelle classi di utilità di `Defects4J`, che fanno largo uso di annotazioni e generics: senza questi gruppi la quasi totalità dei metodi sovrascritti o annotati verrebbe ignorata dall'estrattore.

Le Categorie di Contesto Statico

Il contesto statico iniettato nel prompt si articola in quattro categorie, ciascuna prodotta da una funzione privata dedicata e fornita al modello come vincolo esplicito.

- **Campi della classe.** I campi di istanza e statici rivelano al modello lo stato interno disponibile, evitando che lo ricostruisca dalla propria memoria parametrica. L'estrazione filtra i falsi positivi derivanti da parole chiave Java (`class`, `interface`, `return`).
- **Metodi pubblici della classe.** Le firme complete (modificatori, tipo di ritorno, nome, parametri), inclusi i costruttori pubblici e protetti, garantiscono che il modello invochi solo metodi realmente esistenti e accessibili; una deduplicazione evita che metodi ripetuti nelle classi anonime inquinino il prompt.
- **Tipi importati.** Gli `import` espongono il namespace dei tipi utilizzabili, impedendo proposte di classi non importate, e segnalano la versione delle dipendenze in uso (es. `commons.lang` contro la variante `lang3`), orientando verso costrutti compatibili.
- **Variabili locali nel metodo difettoso.** È la categoria a maggior specificità contestuale: limitata al corpo del metodo coinvolto nel difetto, previene i conflitti di scoping costringendo il modello a riutilizzare variabili già dichiarate, anziché re-inizializzarle o introdurre identificatori in collisione.

Oltre alle quattro categorie di vocabolario, l'*Ingredient Forge* rileva anche le **località gemelle** (`sibling_occurrences`): altre posizioni nella stessa classe in cui ricorre *verbatim* l'espressione che provoca il crash. Quando lo stesso difetto latente è replicato in metodi "specchio", come nel bug JacksonDatabind-80, dove era stata corretta una sola delle due occorrenze una patch limitata alla riga di crash resterebbe parziale. Per questo le occorrenze gemelle non confluiscono nel blocco di vocabolario, ma in una sezione di avviso dedicata (*Suspect twin locations*) che invita il modello a correggerle tutte.

La Struttura Dati Ingredients

Il contratto di dati è formalizzato nella struttura `Ingredients`.

```
@dataclass
class Ingredients:
    """Contesto statico raccolto proattivamente da
       IngredientForge.
    Iniettato nel prompt del LLM per evitare che allucini.
    """
    class_fields: List[str] = field(default_factory=list)
    """Dichiarazioni dei campi della classe target."""
```

```
    public_methods: List[str] = field(default_factory=list)
    """Firme dei metodi pubblici/protetti della classe target.
       """

    imported_types: List[str] = field(default_factory=list)
    """Tipi fully-qualified importati nel file sorgente."""

    local_variables: List[str] = field(default_factory=list)
    """Variabili visibili nello scope della riga di errore."""

    sibling_occurrences: List[str] = field(default_factory=list
       )
    """Occorrenze dell'espressione di crash altrove nella
       stessa
    classe (potenziali localita gemelle da correggere insieme).
       """
```

La struttura è volutamente "piatta" (liste di stringhe a un solo livello), in contrasto con AST o oggetti JSON annidati. Le stringhe sono il formato nativo dei token su cui operano gli LLM: fornendole pre-formattate si elimina una serializzazione costosa e soggetta a perdita di informazioni, massimizzando efficienza e intelligibilità dell'input.

Esempio: gli `Ingredients` estratti per Csv-4. Applicando `collect()` alla classe `CSVParser`, il blocco di contesto statico iniettato nel prompt assume la forma seguente:

```
Campi:      CSVFormat format, Lexer lexer, List<String> record
   ,
            long recordNumber, Token reusableToken
Metodi:     public List<CSVRecord> getRecords(), public
   boolean isClosed(),
            public long getCurrentLineNumber(), public
               Iterator<CSVRecord> iterator(),
            public CSVParser(final Reader reader, final
               CSVFormat format)
Import:     java.util.LinkedHashMap, java.util.Map, java.util.
   ArrayList, ...
Locali:     hdrMap, formatHeader, header   # nel corpo di
   initializeHeader()
```

Sono questi gli "strumenti reali" con cui il modello costruisce la patch: in particolare, la presenza di `java.util.LinkedHashMap` fra gli import rende lecita la riparazione vincente `new LinkedHashMap<>(this.headerMap)`, mentre la lista dei metodi pubblici impedisce al modello di invocare API inesistenti nella classe.

Il Meccanismo di Iniezione: dalla Struttura al Prompt

L'ultimo passaggio inietta i dati raccolti nel prompt utente tramite la funzione helper `_format_list()`.

```
def _format_list(items: list, max_items: int = 20) -> str:
    """Format a list as a comma-separated string, truncating if
        needed."""
    if not items:
        return "(none available)"
    truncated = items[:max_items]
    result = ", ".join(f"`{item}`" for item in truncated)
    if len(items) > max_items:
        result += f" ... (+{len(items) - max_items} more)"
```

Il meccanismo concentra tre decisioni di prompt engineering:

- **Troncamento a 20 elementi** per lista, bilanciamento empirico tra completezza e *context pressure*: iniettare centinaia di metodi saturerebbe la finestra di contesto, distogliendo l'attenzione del modello dalle informazioni diagnostiche primarie.
- **Backtick Markdown** attorno a ogni elemento come *instruction reinforcement*: gli LLM associano i backtick a identificatori letterali e tendono a trattarli come entità intoccabili, riducendo il rischio di parafrasi o mutazioni sintattiche.
- **Fallback "(none available)"** per le liste vuote: una stringa esplicita anziché uno spazio vuoto evita che il modello interpreti il campo come un errore di formattazione o un dato mancante.

Per Csv-4, l'iniezione produce una sezione del prompt della forma:

```
### Available API (Repair Ingredients)
**Class fields:**
`CSVFormat format`, `Lexer lexer`, `List<String> record`, ...
**Public methods:**
`public List<CSVRecord> getRecords()`, `public boolean isClosed()`,
`public CSVParser(final Reader reader, final CSVFormat format)`, ...
**Imported types:**
`java.util.LinkedHashMap`, `java.util.Map`, `java.util.ArrayList`, ...
```

Tramite questa iniezione, il prompt si ancora all'istanza concreta del bug, vincolando lo spazio decisionale del modello alla realtà del codice sorgente.

3.4.3 Il Paradigma Chain-of-Thought: `spec_builder.py`

Il modulo `spec_builder.py` è l'infrastruttura di ragionamento della Fase 2. Dopo la raccolta del contesto statico, il sistema deve tradurre le informazioni diagnostiche in un piano d'azione coerente, affidandosi al paradigma Chain-of-Thought (CoT) per forzare la scomposizione analitica del problema.

Fondamento Teorico del Chain-of-Thought Prompting

Il Chain-of-Thought prompting, formalizzato da Wei et al. [42], si basa su un'intuizione semplice: istruire il modello a produrre una catena di ragionamento intermedio prima della risposta finale migliora la qualità dell'output. La probabilità di generare un token corretto al passo t aumenta se preceduto da un ragionamento esplicitato nei token antecedenti [42].

Nell'APR il paradigma è imprescindibile: chiedere a un LLM di generare direttamente una patch da un report di errore equivale a forzare un problema "multi-hop" in un singolo salto cognitivo, fondendo quattro operazioni (comprendere il difetto, dedurre il comportamento corretto, isolare la differenza minima, tradurla in sintassi Java). Costretto a eseguirle in parallelo, il modello tende a scorciatoie cognitive, pattern di fix generici come un null-check casuale, rinunciando a una soluzione mirata per l'istanza.

La validità del CoT nell'APR è dimostrata da sistemi come `ThinkRepair` [50], dove l'anteposizione di un passo di ragionamento ("think step by step") incrementa il tasso di successo su `Defects4J` di circa 15 punti percentuali. `HybridRepair` radicalizza l'approccio: anziché una richiesta di ragionamento libera, costringe il modello a rispondere a tre interrogativi puntuali in formato XML rigoroso, rendendo ogni fase parzialmente verificabile in modo automatico.

Architettura del System Prompt di SpecBuilder

Il prompting di `spec_builder.py` sfrutta l'architettura a messaggi dell'API OpenAI, separando un *system prompt* (che definisce la persona del modello e i suoi confini comportamentali) da uno *user prompt* (che inietta i dati dell'istanza). Il system prompt di `SpecBuilder` è il seguente:

```
SPEC_SYSTEM_PROMPT = """\
You are an expert Java bug analyst. Your task is to reason
   about a bug \
BEFORE writing any code. You must provide a precise natural-
   language \
specification of the fix, NOT the code itself.

Respond ONLY in the structured format below no code, no diffs,
   no patches.\
"""
```

Pur composto da poche parole, il `SPEC_SYSTEM_PROMPT` concentra tecniche di *instruction reinforcement* [43]. Il maiuscolo su "BEFORE" ("reason about a bug BEFORE writing any code") spezza l'autocompletamento istintivo: essendo gli LLM autoregressivi, un contesto saturo di codice Java li induce a generare subito altro codice [6], e imporre una precedenza temporale disinnesca questo riflesso. I vincoli negativi ("NOT

the code itself", "no code, no diffs, no patches") esplicitano i divieti, fondamentali per reprimere comportamenti eseguiti per inerzia generativa [28]; il token "ONLY" funge da guardrail contro prosa libera e commenti che comprometterebbero il parsing XML deterministico [43].

Il Template del Prompt Utente: Anatomia di SPEC_USER_TEMPLATE

Se il system prompt stabilisce le regole d'ingaggio, il SPEC_USER_TEMPLATE costituisce la mappa operativa in cui il modello deve navigare:

```
SPEC_USER_TEMPLATE = """\
## Bug Analysis Task

**Bug ID:** {bug_id}

### Fault Location
- **Class:** `{class_name}`
- **Line:** {line}
- **Method:**
```java
{method_source}
```

Prolog Causal Analysis (from LogicFL - Semantically
   Grounded)
{causal_analysis}

Repair Directives (from Prolog Rule Analysis)
{repair_directives}

Failing Test(s)
{failing_tests}

Stack Trace
```
{stack_traces}
```

Available API (Repair Ingredients)
Class fields: {class_fields}
Public methods: {public_methods}
Imported types: {imported_types}
Local variables: {local_variables}

Your Task

Answer these three questions precisely and concisely:

1. Flawed Behavior
```

```
What does the buggy method currently do wrong? Be specific
   about the \
exact line and the nature of the defect (e.g., "line 42
   dereferences \
‘result‘ which can be null when the input array is empty").

### 2. Intended Behavior
What SHOULD the method do? Infer this from the test
   expectations and \
any JavaDoc comments. Be specific.

### 3. Minimal Fix
What is the SMALLEST conceptual change that fixes the bug?
   Describe it \
in plain English (e.g., "Add a null check for ‘result‘ before
   line 42, \
returning an empty array if null"). Do NOT write code.

### 4. Fix Locations
For EACH fault location listed above (and each suspect twin
   location), \
state whether the fix must touch it, and why. A fix that
   silences the \
crash point while leaving an upstream cause unaddressed is
   INVALID.

Respond in this exact format:
<flawed_behavior>...</flawed_behavior>
<intended_behavior>...</intended_behavior>
<minimal_fix>...</minimal_fix>
<fix_locations>...</fix_locations>
```

Il template ha una struttura bipartita separata da un delimitatore orizzontale (`---`): un Blocco di Evidenza e un Blocco di Compito.

Il Blocco di Evidenza aggrega le informazioni diagnostiche delle fasi precedenti: Bug ID e Fault Location col sorgente del metodo, la Prolog Causal Analysis e le Repair Directives generate da `LogicFL`, e le tracce empiriche del fallimento (Failing Tests e Stack Trace). In chiusura, la sezione “Available API (Repair Ingredients)” incorpora campi, metodi pubblici, tipi importati e variabili locali estratti dall’Ingredient Forge. Il loro posizionamento in fondo al blocco non è casuale: collocare le API reali subito prima del Task sfrutta il *recency bias* dei Transformer [54], mitigando l’effetto *Lost in the Middle* [27] e riducendo il rischio che l’attenzione divaghi verso librerie allucinate.

Il Blocco di Compito, oltre il separatore, chiede al modello di rispondere a quattro domande specifiche:

- *Flawed Behavior*: cosa fa di sbagliato il metodo, con la riga esatta;
- *Intended Behavior*: cosa dovrebbe fare, inferendolo dalle aspettative dei test o dai JavaDoc;

- *Minimal Fix*: la più piccola alterazione logica che risolve il problema, in linguaggio naturale e senza scrivere codice;
- *Fix Locations*: per ogni località della catena causale (e ogni possibile gemella) se la patch deve toccarla e perché, una correzione che zittisce il crash ma lascia irrisolta una causa a monte è invalida.

La separazione tramite delimitatori impedisce la sovrapposizione tra contesto e istruzioni [43]; l'incapsulamento della risposta in tag pseudo-XML (`<flawed_behavior>`, ecc.) applica il *format manager pattern* [43], essenziale per il parsing deterministico a valle.

Per Csv-4, il modello produce la `RepairSpec` seguente (riportata nella forma a tag elicitata dal template; nel prompt della Fase 3 viene poi riformattata in markdown):

```
<flawed_behavior>
A NullPointerException occurs in getHeaderMap() at line 288
   when it
creates a new LinkedHashMap from the headerMap field, which is
   null.
The initializeHeader() method (line 351) assigns null to hdrMap
    under
certain conditions, and this null propagates to headerMap via
   the
constructor (line 244). Dereferencing headerMap without a null
   check
is the direct cause of the crash.
</flawed_behavior>
<intended_behavior>
getHeaderMap() should return a copy of the headerMap field when
    it is
initialized. If headerMap is null, the method should handle
   this
gracefully -- returning null (if allowed by the class contract)
    or an
empty map -- instead of crashing, matching the test
   expectations.
</intended_behavior>
<minimal_fix>
Add a null check in getHeaderMap() at line 288 so the null case
    is
handled gracefully; optionally adjust initializeHeader() (line
   351)
to avoid propagating null. The change is minimal yet sufficient
    to
prevent the NullPointerException.
</minimal_fix>
<fix_locations>
Location 1 (CSVParser:288) -- must change: crash point, add the
    null
guard here. Location 2 (CSVParser:351) -- optional: could stop
initializeHeader() from returning null. Location 3 (CSVParser
   :244)
```

```
-- no change needed: the constructor correctly assigns the
   result.
</fix_locations>
```

La Costruzione Dinamica del Prompt: _build_spec_prompt()

Una volta definiti i template statici e raccolti tutti gli ingredienti necessari, il sistema deve fondere queste eterogenee fonti di informazione in un unico prompt testuale coerente. Questa delicata operazione di assemblaggio è demandata alla funzione interna _build_spec_prompt().

```
def _build_spec_prompt(
    fault_report: FaultReport,
    ingredients: Ingredients,
) -> str:
    primary = fault_report.locations[0] if fault_report.
       locations else None

    if fault_report.grounded_chains:
        from fault_oracle.prolog_grounding import
           format_grounded_chains
        causal_analysis = format_grounded_chains(fault_report.
           grounded_chains)
        directives = []
        for gc in fault_report.grounded_chains:
            if gc.repair_directive:
                directives.append(f"- {gc.repair_directive}")
        repair_directives = "\n".join(directives) if directives
            else "(none)"
    else:
        causal_analysis = "\n".join(fault_report.causal_chains)
            or "(none)"
        repair_directives = "(no rule-based directives
           available)"

    return SPEC_USER_TEMPLATE.format(
        bug_id=fault_report.bug_id,
        class_name=primary.class_name if primary else "unknown"
           ,
        line=primary.line if primary else 0,
        method_source=primary.method_source if primary else "(
           not available)",
        causal_analysis=causal_analysis,
        repair_directives=repair_directives,
        failing_tests="\n".join(
            f"- `{t['class']}::{t.get('method', '?')}`"
            for t in fault_report.failing_tests
        ) or "(none)",
        stack_traces=fault_report.stack_traces[:3000] or "(none
           )",
        class_fields=_format_list(ingredients.class_fields),
```

```
        public_methods=_format_list(ingredients.public_methods)
            ,
        imported_types=_format_list(ingredients.imported_types)
            ,
        local_variables=_format_list(ingredients.
            local_variables),
```

La funzione non è una semplice interpolazione di stringhe: incarna il principio della *graceful degradation*. Le catene causali di `LogicFL` sono idealmente arricchite dalla pipeline di Semantic Grounding, ma l'analisi statica può fallire (sorgente non localizzabile, Tree-sitter in errore su sintassi malformata, output LogicFL imprevisto). Anziché bloccare la pipeline, la funzione ripiega automaticamente sulle catene causali Prolog grezze: il prompt risulta meno ricco, ma l'esecuzione prosegue fornendo comunque all'LLM una traccia logica essenziale.

Completato l'assemblaggio, `SpecBuilder` invoca il modello tramite il client Azure OpenAI con temperatura molto bassa (`temperature=0.1`). A differenza dei task creativi, dove valori prossimi a 1.0 massimizzano la diversità [38], nel ragionamento deduttivo la variabilità è controproducente: il sistema cerca la spiegazione fattualmente più probabile dalle prove empiriche. Una temperatura di 0.1 riduce la varianza del campionamento, massimizza l'aderenza ai fatti, mitiga le allucinazioni [21] e garantisce la riproducibilità della specifica.

Il Parsing XML della Risposta e la Struttura RepairSpec

La risposta del modello deve essere decodificata in una struttura dati. Il sistema impone l'incapsulamento delle deduzioni in un formato XML semi-strutturato con tre tag (`<flawed_behavior>`, `<intended_behavior>`, `<minimal_fix>`). La scelta dell'XML rispetto al JSON è motivata dalla robustezza: la grammatica JSON è severa e forzare un LLM a produrre JSON valido con testo libero multiparagrafo comporta un rischio elevato di errori di parsing [56]. L'XML, al contrario, tollera nativamente testo discorsivo e ritorni a capo [3], ed è ampiamente rappresentato nel corpus di pre-training (HTML, file di configurazione), risultando più affidabile. Per estrarre il contenuto dei tag, `spec_builder.py` implementa un parser a espressioni regolari.

La funzione interna di parsing, `_parse_spec_response()`, si avvale di una regex: `rf"<{tag}>\s*(.*?)\s*</{tag}>"` accoppiata al flag di compilazione `re.DOTALL`.

```
def _parse_spec_response(response: str) -> RepairSpec:
    import re

    def extract(tag: str) -> str:
        m = re.search(rf"<{tag}>\s*(.*?)\s*</{tag}>", response,
            re.DOTALL)
        return m.group(1).strip() if m else ""

    return RepairSpec(
        flawed_behavior=extract("flawed_behavior"),
        intended_behavior=extract("intended_behavior"),
        minimal_fix=extract("minimal_fix"),
        fix_locations=extract("fix_locations"),
```

La regex è progettata per tollerare le imperfezioni stilistiche del modello. Il flag `re.DOTALL` consente al punto di catturare anche i newline, così da estrarre blocchi distribuiti su più righe, mentre i gruppi `\s*` ai bordi assorbono spaziature e ritorni a capo spuri. Il dettaglio cruciale è il quantificatore *lazy* in `(.*?)`: un `.*` "ingordo" catturerebbe tutto fino all'ultimo tag di chiusura della stringa, e in presenza di tag duplicati distruggerebbe la frammentazione della risposta, mentre la versione lazy si ferma al primo tag di chiusura corrispondente. Il `.strip()` finale rimuove le spaziature marginali residue, e il risultato viene incapsulato nella struttura `RepairSpec`.

```
@dataclass
class RepairSpec:
    """Natural-language specification produced by Phase 2 (
       SpecReason).
    The LLM reasons about the bug *before* generating any code.
    """
    flawed_behavior: str = ""
    """What the buggy method currently does wrong."""

    intended_behavior: str = ""
    """What the method *should* do, inferred from tests +
       JavaDoc."""

    minimal_fix: str = ""
    """Conceptual description of the smallest change that fixes
        the bug."""

    fix_locations: str = ""
    """Verdetto per-localita: quali punti della catena causale
       la
    patch deve toccare e perche."""

    confidence: float = 0.0
    """Auto-valutazione di affidabilita prodotta dal SpecCritic
        (0.0-1.0)."""
```

La `RepairSpec` è un oggetto immutabile, artefatto di trasferimento verso le fasi successive. Mantenere i campi diagnostici in linguaggio naturale anziché in linguaggi di specifica formale (Alloy, JML) è strategico: il linguaggio naturale è il medium ottimale per comunicare l'intento di correzione al `PatchAgent` della Fase 3. Ai quattro campi testuali prodotti da `SpecBuilder` si aggiunge il `confidence`, un punteggio numerico popolato in seguito dal `SpecCritic`. Applicato alla risposta del modello per Csv-4 (mostrata nella Sezione 3.4.3), il parser popola i campi con la diagnosi del crash a riga 288, il comportamento atteso, la correzione minima e il verdetto per-località: è questo artefatto a cristallizzare la comprensione del problema, trasformando dati di debug grezzi in un piano di riparazione leggibile e pronto per essere tradotto in codice.

3.4.4 SpecCritic: Validazione Autonoma della Specifica di Riparazione

A valle di `SpecBuilder`, il modulo `spec_critic.py` valuta, critica ed eventualmente emenda le deduzioni della fase precedente.

Il Problema della Coerenza Interna nelle Specifiche Generate da LLM

Una `RepairSpec` generata in CoT può essere internamente coerente e plausibile ma fattualmente incongruente con le prove empiriche [44, 10]. Ad esempio, il modello potrebbe identificare correttamente che `r` è `null` ma attribuirne l'origine a un metodo errato (confondendo firme simili), oppure diagnosticare correttamente il difetto ma proporre una soluzione sovra-ingegnerizzata (refactoring strutturale anziché un singolo controllo condizionale). Per arginare queste derive, `spec_critic.py` implementa un meccanismo di auto-critica ispirato alla Constitutional AI [4], qui declinato in forma alleggerita: un singolo turno di validazione mirato a intercettare gli errori logici o fattuali più evidenti prima della generazione del codice.

Il System Prompt del SpecCritic e la Separation of Concerns

L'auto-critica si fonda sul principio della *separation of concerns* traslato nel prompt engineering: se lo `SpecBuilder` istruiva il modello come "analista", il system prompt dello SpecCritic gli impone il ruolo opposto di "meticulous code review expert", incaricato di convalidare o rigettare il lavoro altrui.

```
CRITIC_SYSTEM_PROMPT = """\
You are a meticulous code review expert. Your job is to
    validate a bug \
analysis produced by another analyst. Check for logical
    consistency, \
```

L'aspetto più sofisticato è il distanziamento psicologico: al modello si chiede di valutare un'analisi "prodotta da un altro analista" (`produced by another analyst`). Non è una finzione narrativa, ma una contromisura tecnica contro la *sycophancy* [35] e il *self-enhancement bias* [55]: i modelli faticano ad auto-correggersi [15] e tendono a validare acriticamente i propri output se ne sono consapevoli autori. Fingendo una terza parte, si abbassa la barriera difensiva del modello, incoraggiando una revisione severa e oggettiva.

Il Template del Prompt del SpecCritic (CRITIC_USER_TEMPLATE)

Il passaggio dal ruolo di generatore a quello di revisore si riflette anche nell'architettura del prompt utente, il `CRITIC_USER_TEMPLATE`, che presenta differenze strutturali e logiche profonde rispetto alla sua controparte nello `SpecBuilder`.

```
CRITIC_USER_TEMPLATE = """\
## Validation Task

An analyst produced the following bug specification. Validate
    it.

**Bug ID:** {bug_id}

### Analyst's Analysis

**Flawed Behavior:**
{flawed_behavior}
```

```
**Intended Behavior:**
{intended_behavior}

**Minimal Fix:**
{minimal_fix}

**Fix Locations (per-location verdict):**
{fix_locations}

### Ground Truth (from test execution)

**Fault Locations (causal chain from fault localization):**
{fault_locations}

**Stack Trace:**
```
{stack_traces}
```

**Failing Tests:**
{failing_tests}

---

## Your Task

1. Is the "Flawed Behavior" description consistent with the
   stack trace?
2. Is the "Intended Behavior" plausible given the test
   expectations?
3. Is the "Minimal Fix" truly minimal, or does it over-engineer
    the solution?
4. COVERAGE CHECK: does the "Fix Locations" verdict account for
    EVERY fault
   location of the causal chain? Patching only the crash point
      while an
   upstream cause stays broken cures the symptom, not the
      disease.
5. Rate your confidence that this specification would lead to a
    correct fix (0.0-1.0).

If the analysis is correct, respond with the SAME content.
If corrections are needed, provide the corrected version.

Respond in this exact format:
<flawed_behavior>...</flawed_behavior>
<intended_behavior>...</intended_behavior>
<minimal_fix>...</minimal_fix>
<fix_locations>...</fix_locations>
<confidence>0.X</confidence>
"""
```

‘‘‘

Il primo elemento di rottura è la sezione “Ground Truth”, che raggruppa le località di fault della catena causale, lo stack trace e i test falliti dall’esecuzione dinamica: la designazione segnala al modello che sono fatti empirici incontrovertibili, da far prevalere sulle inferenze testuali in caso di conflitto. A seguire, il template sostituisce le domande aperte con cinque criteri di validazione numerati che impediscono un’approvazione superficiale:

1. **Coerenza logica**: la descrizione del difetto è coerente con la “Ground Truth”?
2. **Plausibilità fattuale**: l’intento correttivo è supportato dalle aspettative dei test?
3. **Minimalità**: la correzione è genuinamente minimale o scade nell’over-engineering? Criterio fondamentale nell’APR, dove le soluzioni invasive comportano un alto rischio di regressioni.
4. **Copertura della catena causale**: il verdetto *Fix Locations* affronta *ogni* località della catena, e non solo il punto di crash. Lasciare irrisolta una causa a monte cura il sintomo, non la malattia.
5. **Confidence Score**: un giudizio quantitativo sulla robustezza della specifica.

Infine, un’istruzione condizionale governa l’output: se la specifica è corretta, il modello restituisce l’XML identico; se individua falle, genera direttamente l’XML emendato. Questa biforcazione evita parafrasi stilistiche inutili e rende la critica immediatamente azionabile, arricchita dal tag `<confidence>`, un float in [0.0, 1.0] che quantifica la fiducia del critico nel fix.

Il Parsing della Risposta e il Meccanismo di Fallback

La trasformazione dell’output testuale del revisore in dati strutturati e azionabili è affidata alla funzione interna `_parse_critic_response()`.

```
def _parse_critic_response(response: str) -> tuple[RepairSpec,
    float]:
    import re

    def extract(tag: str) -> str:
        m = re.search(rf"<{tag}>\s*(.*?)\s*</{tag}>", response,
            re.DOTALL)
        return m.group(1).strip() if m else ""

    confidence_str = extract("confidence")
    try:
        confidence = float(confidence_str)
    except (ValueError, TypeError):
        confidence = 0.5  # default if parsing fails

    return RepairSpec(
        flawed_behavior=extract("flawed_behavior"),
        intended_behavior=extract("intended_behavior"),
        minimal_fix=extract("minimal_fix"),
```

```
        fix_locations=extract("fix_locations"),
        confidence=confidence,
```

Da un punto di vista architetturale, questa funzione non si discosta dalla logica basata su espressioni regolari già impiegata nello `SpecBuilder`, ma la estende per accogliere i tag aggiuntivi del critico: `<fix_locations>`, trattato come gli altri campi testuali, e soprattutto `<confidence>`, che richiede un float in [0.0, 1.0] e una gestione rigorosa: il modello potrebbe omettere il tag, inserirvi testo discorsivo (es. `<confidence>High (0.9)</confidence>`) o restituire formati non convertibili. Di fronte a un `ValueError/TypeError` nel casting, la funzione adotta un fallback conservativo a 0.5. Il valore mediano è calibrato: 0.0 penalizzerebbe ingiustamente una specifica magari ineccepibile per un mero errore di formato, 1.0 le attribuirebbe un'affidabilità non verificata. Lo 0.5 segnala un'incertezza formale dovuta a un intoppo tecnico; la funzione coordinatrice emette inoltre un log ("`WARNING:` low confidence") quando il valore scende sotto soglia.

La robustezza è ulteriormente garantita dalla funzione `validate()`: se l'LLM restituisce tag essenziali (`<flawed_behavior>`, `<minimal_fix>`) vuoti o malformati, anziché scartare l'artefatto o inoltrare un oggetto vuoto, il sistema recupera i campi originari dello `SpecBuilder` e li innesta al posto di quelli vuoti. Nell'APR un contesto parziale vale più di un contesto assente: conservare il ragionamento originale evita che il fallimento di un singolo controllo paralizzi l'intera pipeline.

Il Flusso di Correzione Iterativa

A differenza della Constitutional AI completa, che prevede cicli multi-iterativi fino a convergenza, `HybridRepair` applica una sola critica per ogni specifica, con al più una ricostruzione condizionata dalla confidence (cfr. Sezione 3.4.5): non un ciclo aperto fino a convergenza, ma al massimo due passaggi di build e critica. La scelta deriva da vincoli ingegneristici: ogni chiamata a `GPT-4o` introduce latenza (2–10 s) e costo per token, e cicli a tre o più iterazioni aggraverebbero i tempi della Fase 2 con un ROI computazionale sfavorevole [9]; in assenza di feedback esterni, le iterazioni successive alla prima non garantiscono un miglioramento proporzionale al costo [39]. Operativamente, una singola iterazione è sufficiente a intercettare le classi di errore più comuni (inconsistenze con lo stack trace, over-engineering, errata localizzazione). Difetti più profondi (es. semantiche di concorrenza) richiederebbero validazione formale o oracoli esterni [36], fuori dalla portata di ulteriori iterazioni introspettive del medesimo modello.

3.4.5 Integrazione nella Pipeline: il Contratto tra Fase 2 e Fase 3

La Fase 2 si conclude producendo un oggetto `RepairSpec` validato, accompagnato dal set di `Ingredients` raccolto. La transizione verso la generazione del codice è governata dall'orchestratore `repair_bug.py`, nel momento in cui viene istanziato il `PatchAgent`, motore della Fase 3.

```
# Frammento da repair_bug.py - transizione Fase 2 -> Fase 3
agent = PatchAgent(
    fault_report=fault_report,
```

```
    repair_spec=spec,           # <- RepairSpec validata dalla
       Fase 2
    ingredients=ingredients,    # <- Ingredients raccolti dall'
       Ingredient Forge
    client=client,
)
```

Il costruttore riceve una triade di oggetti tipizzati: il `FaultReport` (eredità diagnostica della Fase 1), la `RepairSpec` validata e gli `Ingredients` statici. Questa firma definisce un contratto software tra i due macro-moduli, separando nettamente la strategia dagli strumenti: la `RepairSpec` descrive in linguaggio naturale *cosa* fare (comportamento difettoso, intento corretto, Minimal Fix) senza vincolarsi a implementazioni premature, mentre gli `Ingredients` descrivono *con quali strumenti* operare, fornendo l'inventario delle API reali (campi, metodi, variabili locali, dipendenze). Il `PatchAgent` riceve così un duplice binario di guida, direzione semantica dalla `RepairSpec` e contenimento sintattico dagli `Ingredients`, trasformandosi da decisore incerto a esecutore vincolato.

Il valore `spec.confidence` calcolato durante l'auto-critica non è un semplice dato diagnostico, ma governa un meccanismo di salvaguardia all'interno della Fase 2. L'orchestratore `repair_bug.py` confronta la confidence con una soglia (`LOW_CONFIDENCE = 0.4`): se il critico si dichiara poco sicuro (`confidence < 0.4`), la specifica viene *ricostruita una volta* con una temperatura più alta (`0.3` anziché `0.1`), così da elicitare un ragionamento alternativo; la nuova bozza viene a sua volta validata, e tra le due si conserva quella con confidence maggiore. La `RepairSpec` che giunge al `PatchAgent` è dunque la migliore di al più due tentativi, e il suo punteggio finale viene riportato nell'output diagnostico (log "`RepairSpec` ready (confidence: X.XX)").

In sintesi

La Fase 2 trasforma la diagnosi grezza della Fase 1 in un piano di riparazione operativo: l'*Ingredient Forge* ancora il modello al vocabolario reale della classe, lo `SpecBuilder` forza un ragionamento Chain-of-Thought prima del codice, e lo `SpecCritic` ne valida la coerenza contro le prove empiriche. Il risultato è una `RepairSpec` validata, affiancata dagli `Ingredients`: insieme offrono al `PatchAgent` della Fase 3 un duplice binario di guida, direzione semantica e contenimento sintattico, su cui generare e verificare la patch.

3.5 Fase 3 - PatchAgent: la FSM e il meccanismo di retry

La terza e ultima fase dell'architettura `HybridRepair`, implementata dal modulo `PatchAgent`, costituisce il nucleo esecutivo del sistema.

Le tre fasi si ripartiscono tre domande distinte. La Fase 1 (`FaultOracle`) localizza il difetto e ne individua le cause, rispondendo a *dove* si trova l'errore e *perché* il programma fallisce. La Fase 2 (`SpecReason`) astrae queste informazioni in una specifica concettuale di riparazione, rispondendo a *cosa* debba essere modificato. Il `PatchAgent` cala questi costrutti nel codice sorgente, rispondendo a *come* il codice Java vada concretamente alterato.

Questa traduzione da specifica concettuale a modifica concreta non è banale: il `PatchAgent` deve garantire che la modifica soddisfi tre criteri:

1. **Compilazione:** deve compilare senza introdurre errori sintattici o di tipizzazione;
2. **Correttezza Semantica:** deve risolvere il difetto semantico facendo passare i test della suite precedentemente falliti;
3. **Preservazione Funzionale:** deve preservare integralmente il comportamento corretto preesistente del sistema, evitando l'introduzione di regressioni.

Per affrontare questa sfida, il `PatchAgent` introduce un duplice contributo architetturale:

- **Il ciclo di riparazione iterativo basato su FSM:** il cuore dell'agente è governato da una Macchina a Stati Finiti (*Finite State Machine*, FSM) implicita che dirige un agente LLM (con un toolkit di sei strumenti) attraverso quattro macro-fasi sequenziali: `EXPLORE`, `PATCH`, `COMPILE` e `TEST`. Un meccanismo di *retry* adattivo, con schedule di temperatura dinamici e memoria conversazionale persistente, permette all'agente di imparare dai propri fallimenti.
- **La riparazione strutturale basata su Tree-sitter:** abbandonando gli approcci testuali (regex, diff, sostituzioni di stringhe), il sistema manipola direttamente il CST (*Concrete Syntax Tree*), garantendo che ogni modifica sia corretta per costruzione dal punto di vista posizionale e sintattico, indipendentemente dallo stile del progetto target.

3.5.1 Il Ruolo del PatchAgent nell'Architettura HybridRepair

Prima di operare, il `PatchAgent` riceve tre artefatti prodotti dalle fasi precedenti, formalizzati nei dataclass di `shared/models.py`:

`FaultReport` **(Fase 1):** la mappa del problema: posizioni del fault (da `LogicFL`), sorgente del metodo difettoso estratto via Tree-sitter (`ast_extractor.py`), catene causali ancorate (da `prolog_grounding.py`) e test falliti con stack trace.

`RepairSpec` **(Fase 2):** il piano d'azione: descrizione in linguaggio naturale del comportamento difettoso, di quello corretto atteso e della modifica minimale, validata dal `SpecCritic` con un punteggio di confidenza.

`Ingredients` **(Fase 2):** il materiale da costruzione: il vocabolario API reale della classe difettosa (campi, metodi, tipi importati, variabili locali) estratto da `ingredient_forge.py`. Di questo vocabolario il `PatchAgent` pre-carica nel prompt solo i campi e i metodi pubblici, recuperando import, variabili locali e firme esatte su richiesta tramite i propri strumenti (cfr. Sezione 3.5.2).

L'orchestratore `repair_bug.py` raccoglie questi tre elementi e istanzia il `PatchAgent`:

```
agent = PatchAgent(
    fault_report=fault_report,
    repair_spec=repair_spec,
    ingredients=ingredients
)

repair_result = agent.run()
```

L'invocazione di `agent.run()` avvia il ciclo di riparazione autonomo e restituisce un oggetto `RepairResult`, che aggrega tutti i tentativi effettuati, identifica il tentativo vincente e conserva lo storico degli artefatti intermedi (conversazioni LLM, diff, output del compilatore). Il flusso governato da `run()` è schematizzato in Figura 3.5.

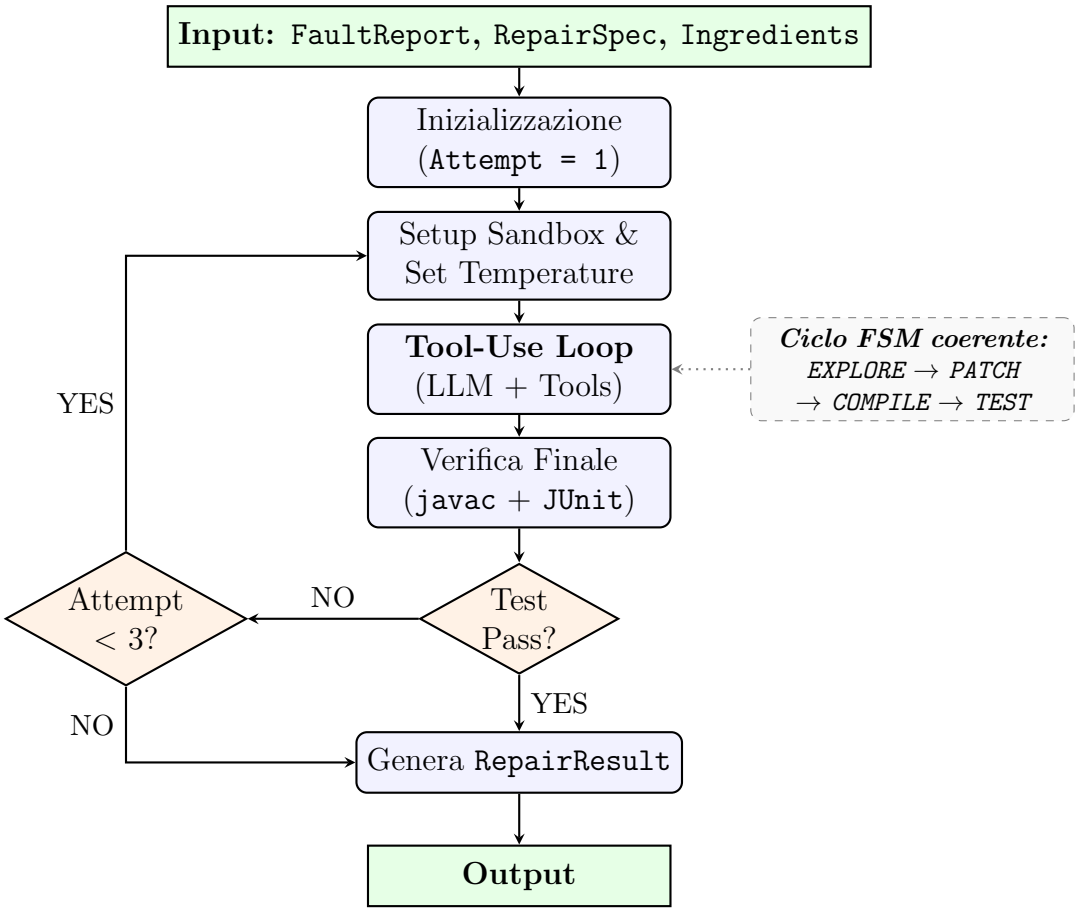

Figura 3.5: Flusso complessivo del ciclo di esecuzione iterativo e adattivo gestito all'interno di `PatchAgent.run()`.

Spiegazione del Flusso di Esecuzione

Il diagramma illustra la natura iterativa e auto-correttiva del `PatchAgent`. All'inizio di ognuno dei (massimo) tre tentativi, il sistema crea una copia isolata del codice difettoso in un sandbox dedicato (`pipeline_results/<bug_id>/attempt_k/patched_source/`), per evitare che le modifiche di un tentativo inquinino i successivi, e imposta la temperatura del modello secondo uno schedule adattivo (0.1, 0.2, 0.4) che ne aumenta la "creatività" a fronte di fallimenti ripetuti.

Preparato l'ambiente, l'esecuzione entra nel *Tool-Use Loop*, dove il modello e il toolkit interagiscono dinamicamente: è qui che prende vita la FSM, guidando l'agente a esplorare il codice, generare e applicare una patch e testarla. L'agente dispone di un budget di massimo 9 round di interazione con i tools.

Usciti dal loop, il sistema procede con una verifica finale deterministica: invoca il compilatore (`javac`) e il framework di test (`JUnit`) sul codice nel sandbox. Se tutti i test passano (`PASS`), la riparazione ha successo e il tentativo viene registrato come vincente. Se i test falliscono o il codice non compila, e restano tentativi disponibili, la storia dei fallimenti viene salvata nella memoria conversazionale e iniettata nel prompt successivo per indicare all'LLM cosa evitare; il ciclo riparte con una nuova copia del codice e una temperatura più elevata. Esauriti i tentativi o raggiunto il successo, il sistema aggrega le informazioni in un oggetto `RepairResult`.

3.5.2 Il Ciclo di Vita dell'Agente: la FSM di PatchAgent

Il comportamento del `PatchAgent` è modellato come una Macchina a Stati Finiti (FSM), formalmente una quintupla $M = (Q, \Sigma, \delta, q_0, F)$. Nell'architettura del `PatchAgent` l'insieme degli stati è

$$Q = \{\texttt{EXPLORE}, \texttt{PATCH}, \texttt{COMPILE}, \texttt{TEST}, \texttt{DONE}, \texttt{RETRY}\}$$

con stato iniziale $q_0 = \texttt{EXPLORE}$ e stato finale $F = \{\texttt{DONE}\}$. La funzione di transizione δ determina come l'agente avanza reagendo all'esito delle proprie azioni; la Figura 3.6 ne illustra le transizioni.

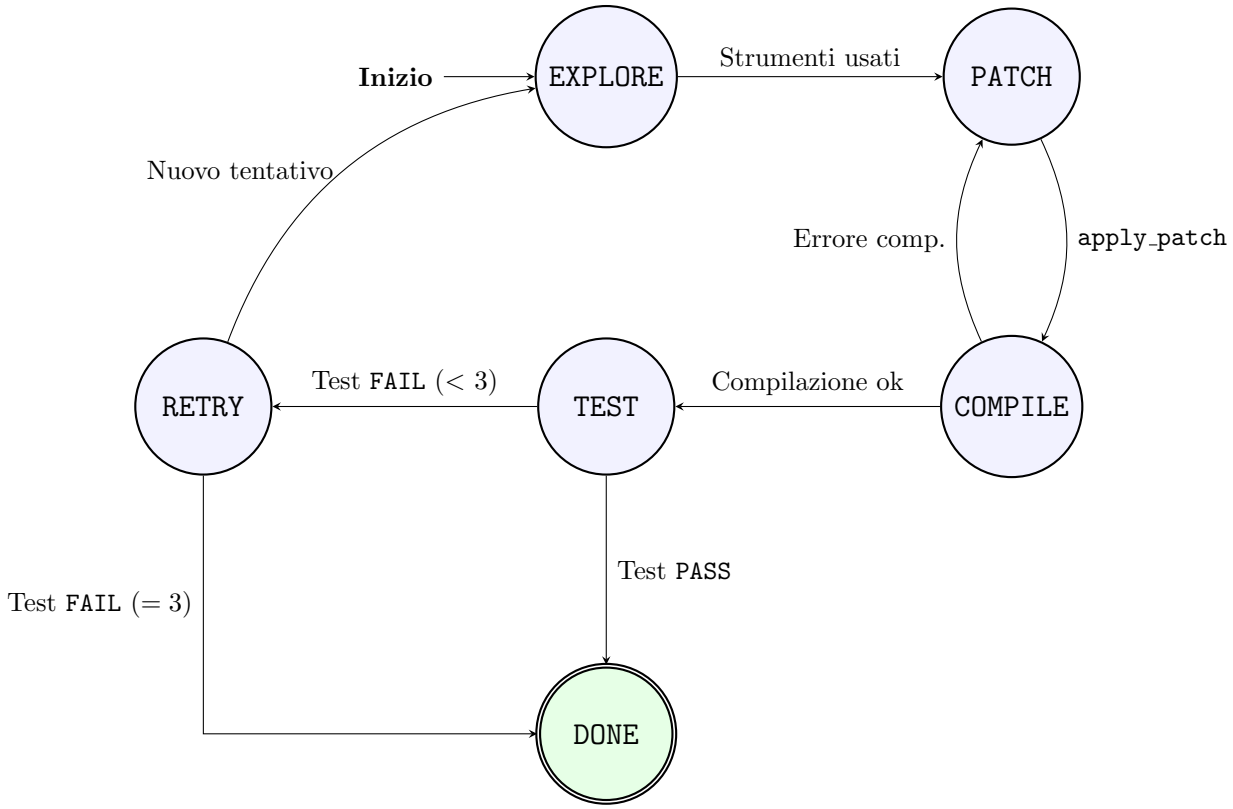

Figura 3.6: Grafo delle transizioni della Macchina a Stati Finiti (FSM) implicita del `PatchAgent`.

Il Concetto di FSM Emergente

Questa FSM non è implementata in modo tradizionale: nel codice non esiste una classe con un attributo `state` esplicito né una tabella di transizioni con `if/switch`. Essa emerge dalla combinazione del *System Prompt* iniettato in GPT-4o e dal loop di controllo di `PatchAgent.run()` e `AzureClient.chat_with_tools()`. Come dimostrato da sistemi come `RepairAgent` e `ChatRepair` [46, 5], i modelli GPT-4 aderiscono a workflow multi-step complessi tramite prompting strutturato, transitando autonomamente tra gli stati e selezionando lo strumento appropriato in ciascun round. La coreografia è diretta dal seguente prompt di sistema:

```
AGENT_SYSTEM_PROMPT = """\
You are an expert Java bug repair agent. Your task is to fix a
    bug in a \
Defects4J Java project using the tools available to you.

## Workflow
1. **EXPLORE**: Read the buggy code, search for related symbols
    , check method signatures.
2. **PATCH**: Apply a fix using `apply_ast_patch`. Provide the
    COMPLETE corrected method.
3. **COMPILE**: Check compilation with `compile_check`.
4. **TEST**: Verify with `run_tests`.

## Rules
- ALWAYS use `apply_ast_patch` to apply fixes. Never output raw
     diffs.
- ALWAYS compile before testing.
- Use REAL API methods only. Call `get_method_signature` to
    verify before using an API.
- If compilation fails, read the error, fix the issue, and try
    again.
- If tests fail, analyze the stack trace and try a different
    approach.
- Keep fixes MINIMAL. Prefer the smallest change that fixes the
     bug.
- Do NOT add unnecessary null checks or try/catch blocks.
...
"""
```

Analisi delle Regole Fondamentali del Prompt

Ogni elemento dell'`AGENT_SYSTEM_PROMPT` vincola l'output del modello, trasformando un LLM *general-purpose* in un risolutore di bug. Tre regole meritano un'analisi:

La Sequenza Numerata delle Azioni: (`1. EXPLORE` → `2. PATCH` → `3. COMPILE` → `4. TEST`): codifica il grafo di transizione della FSM in linguaggio naturale. I Transformer sono *stateless* e privi di una rappresentazione nativa di un automa, ma nei dati di addestramento una lista numerata è associata a una sequenza causale di azioni [41]: il modello è così indotto a esplorare prima di modificare, in linea con i paradigmi *Reasoning and Acting* [49].

Il Divieto di Diff Testuali: (`"ALWAYS use apply_ast_patch...Never output raw diffs."`): senza vincoli, i modelli tendono a eludere gli strumenti ripiegando su diff testuali nella prosa, soggetti a errori di formattazione e incompatibili con l'applicazione automatizzata [19]. Il *negative constraint* forza l'agente a delegare ogni modifica a `apply_ast_patch` [33]; senza di esso il modello potrebbe descrivere a parole una soluzione senza applicarla, vanificando la fase di `COMPILE`.

Il Vincolo di Minimalità: (`"Keep fixes MINIMAL..."`): recepisce un principio dell'APR [17]: le patch minime hanno minore probabilità di indurre regressioni. Le patch *single-hunk* sono più plausibili e generalizzabili delle *multi-hunk* [45], e privilegiare la minima perturbazione mitiga l'overfitting sui test falliti.

La Fase EXPLORE: Esplorazione Autonoma del Workspace

Nella fase `EXPLORE` l'agente costruisce la comprensione contestuale del difetto prima di alterare il codice, invocando tre strumenti del toolkit: `read_file`, `search_symbol` e `get_method_signature`. La fase potrebbe sembrare ridondante, dato che il prompt iniziale fornisce già il metodo difettoso, la `RepairSpec` e parte degli ingredienti. A differenza del prompt one-shot della Fase 2, che pre-carica tutte e quattro le categorie di vocabolario, il `PatchAgent` riceve un'iniezione volutamente più snella, limitata ai campi e ai metodi pubblici della classe: import, variabili locali e firme esatte non vengono pre-caricati. Essendo l'agente dotato di strumenti, può infatti recuperarli su richiesta solo quando servono, evitando di saturare il contesto con informazioni che potrebbero non essere necessarie. Inoltre, anche le informazioni iniettate sono localizzate attorno all'epicentro del difetto, e due categorie di anomalie rendono insufficiente questo perimetro:

Bug con propagazione cross-method: La causa radice non sempre risiede nel metodo in cui l'eccezione viene sollevata: può nascondersi in un metodo chiamato a cascata o in interfacce implementate da una sottoclasse. In questi casi l'agente deve navigare oltre il file iniziale, leggendo moduli aggiuntivi per mappare il flusso di esecuzione prima di formulare una patch semanticamente risolutiva.

Bug con API ambigue: Anche quando la `RepairSpec` suggerisce una specifica API, la firma esatta può non essere chiara a priori (overload multipli, parametri generici con vincoli). Lo strumento `get_method_signature` permette all'agente di ispezionare le firme reali e disambiguare prima di sintetizzare il codice.

La durata della fase `EXPLORE`, misurata in numero di round di invocazione degli strumenti, non è prefissata: l'agente valuta autonomamente quando ha raccolto informazioni sufficienti per agire. La transizione a `PATCH` è implicita, innescata nel momento in cui l'output del modello include un'invocazione allo strumento `apply_ast_patch`.

La Fase PATCH: Applicazione Chirurgica della Modifica

Nella fase `PATCH` si compie il primo atto *generativo* sul codice: è qui che l'LLM produce concretamente la patch. L'agente, però, non modifica mai il file direttamente; secondo il protocollo di *function calling*, emette una chiamata strutturata allo strumento `apply_ast_patch`. La patch "nasce" interamente in uno degli argomenti di questa chiamata: nel campo `new_body` il modello scrive il codice sorgente completo del metodo corretto (firma, graffe e corpo), mentre gli altri argomenti, `file_path`,

`method_name` e, se serve, `target_line` come *hint* per disambiguare gli overload, indicano *dove* applicarlo. Per il bug `Csv-4`, la chiamata vincente del secondo tentativo è:

```
apply_ast_patch(
    file_path="CSVParser.java",
    method_name="getHeaderMap",
    new_body="""public Map<String, Integer> getHeaderMap() {
    return this.headerMap == null ? null : new LinkedHashMap<>(
       this.headerMap);
}""",
)
```

Ricevuta la chiamata, il *tool executor* esegue `apply_ast_patch` in modo *deterministico*: `ast_applier.py` individua il nodo del metodo nel CST di Tree-sitter e ne sostituisce il *text span* con il contenuto di `new_body`. Generazione (probabilistica, a carico dell'LLM) e applicazione (deterministica, a carico dello strumento) restano così nettamente separate.

In questa fase vige il vincolo più rigoroso del sistema: il modello deve fornire il metodo nella sua interezza (firma, graffe e corpo). Diff unificati o frammenti parziali sono vietati, come ribadito nel prompt di sistema (`"Provide the COMPLETE corrected method"`) e nella definizione dello strumento. Questa scelta, all'apparenza dispendiosa in token, è sorretta da due motivazioni, una cognitiva, lato modello, e una infrastrutturale, lato strumento:

Robustezza della generazione LLM: i diff testuali sono al tempo stesso un *formato* fragile e una *consegna* fuorviante per un LLM. Da un lato, generare righe di contesto esatte, indentazione e conteggi di riga corretti è soggetto ad allucinazioni strutturali [53], e in un ciclo iterativo un contesto imperfetto fa fallire subito l'applicazione. Dall'altro, quando è istruito a produrre "solo la modifica", il modello tende a risposte iper-focalizzate: si concentra sulla riga dell'errore (es. un `if (obj != null)`) e trascura il bilanciamento delle graffe o la rilocazione di variabili dichiarate altrove. Imporre la riscrittura del metodo per intero risolve entrambi i problemi: sposta il compito dalla manipolazione testuale alla sostituzione semantica a livello di AST [29] e mantiene l'intera struttura logica nell'orizzonte di attenzione del modello, con un tasso di successo superiore ai diff riga per riga [46].

Compatibilità con Tree-sitter: L'infrastruttura di manipolazione (`ast_applier.py`) opera sul *Concrete Syntax Tree* (CST) di Tree-sitter, individuando il nodo del metodo (`method_declaration` o `constructor_declaration`) e isolandone il *text span*. Per sostituire quel text span l'input deve essere un metodo Java completo e autonomo: frammenti isolati romperebbero l'albero.

La Fase COMPILE: Feedback Deterministico dal Compilatore

Applicata la patch tramite l'AST, l'agente transita nella fase `COMPILE`, primo "oracolo deterministico" del ciclo. A differenza del giudizio probabilistico dell'LLM, il compilatore `javac` applica esattamente la grammatica del linguaggio ed emette un verdetto binario: la patch è valida sotto il profilo della tipizzazione statica, oppure viene rifiutata.

L'interazione avviene tramite lo strumento `compile_check`; la logica esposta all'LLM, che si appoggia a `sandbox_evaluator.py`, è riassumibile così:

```
def tool_compile_check(context: ToolContext) -> str:
    """Verifica la compilazione del file sorgente modificato.
       """
    result = compile_patched_source(context.patched_dir,
       context.bug_dir)

    if result.is_success:
        return "COMPILATION SUCCESS"
    else:
        # Troncamento diagnostico per preservare il budget di
           token
        error_msg = result.stderr[:3000]
        return f"COMPILATION ERROR:\n{error_msg}"
```

Aspetto cruciale è il troncamento dell'output a 3000 caratteri: `javac`, di fronte a un errore sintattico, produce una cascata verbosa di errori secondari che saturerebbe il *token budget* aggiungendo rumore. Troncando, il sistema fornisce all'agente solo i primi errori (le cause radice), nel formato standard `NomeFile.java:LineaNum: error: ...`, da cui GPT-4o estrae autonomamente il punto del fallimento.

Questo feedback innesca il ciclo di *retry inline* (`PATCH` → `COMPILE` → `PATCH`): se la compilazione fallisce, il messaggio di errore viene reiniettato nel contesto e l'agente corregge la patch nello stesso round con una nuova invocazione ad `apply_ast_patch`. Il micro-ciclo consuma il budget di round (`max_tool_rounds`) ma permette di correggere sviste sintattiche minori senza scartare l'intera strategia.

La Fase TEST: Validazione tramite Suite JUnit

Superata la fase `COMPILE`, la FSM avanza allo stato più critico, `TEST`: se il compilatore garantisce che il codice è eseguibile, JUnit funge da secondo oracolo. È però un oracolo *parziale*: superare la suite non certifica la *correttezza* della patch, ma solo la sua *plausibilità* (*test-adequacy*), poiché una batteria di test finita può essere sovra-adattata (*patch overfitting*) e non copre i comportamenti non testati [32].

L'agente innesca la verifica tramite lo strumento `run_tests`, che si appoggia al sandbox di valutazione:

```
def tool_run_tests(context: ToolContext) -> str:
    """Esegue la suite JUnit per verificare la plausibilita'
       della patch."""
    test_result = evaluate(context.patched_dir, context.
       test_classes)

    if test_result.status == "PASS":
        return "PASS: All tests passed successfully."

    elif test_result.status == "COMPILE_ERROR":
        return f"COMPILE_ERROR:\n{test_result.error_output}"

    elif test_result.status == "TEST_FAIL":
        return (f"TEST_FAIL:\n{test_result.stack_traces}\n"
                f"Previously failing now passing: {test_result.
                   previously_failing_now_passing}")
```

L'esecuzione dei test può condurre a tre scenari di output ben distinti, che forniscono all'agente un feedback semanticamente ricco:

`PASS`: Rappresenta il successo totale. Tutti i test della suite, inclusi quelli che originariamente evidenziavano il bug, sono stati eseguiti senza sollevare eccezioni. In questo scenario, la FSM transita direttamente allo stato `DONE` e la riparazione è considerata vincente.

`COMPILE_ERROR`: Scenario atipico in cui la verifica finale (che ricompila il progetto prima di lanciare JUnit) rileva difetti sintattici residui sfuggiti al precedente `compile_check` isolato: il codice, allo stato attuale, non è eseguibile.

`TEST_FAIL`: La patch compila ma non supera la suite: almeno un test fallisce, segnalando un comportamento errato o incompleto rispetto a ciò che i test verificano. Lo strumento restituisce all'LLM l'intero stack trace delle eccezioni sollevate dai test falliti, da cui l'agente può capire per quale input o in quale condizione logica la patch ha ceduto e riconsiderare il proprio approccio.

Il Meccanismo di Retry: Schedule di Temperatura e Adattamento

Quando la fase `TEST` fallisce, il ciclo non si interrompe: la FSM entra nella fase transizionale di `RETRY`, che prepara un nuovo tentativo. Per evitare di ricadere negli stessi errori, il meccanismo di retry si fonda su tre pilastri: una modulazione adattiva della temperatura del modello, l'isolamento del workspace e l'iniezione della memoria conversazionale dei fallimenti pregressi.

1. Schedule di Temperatura Adattiva

La temperatura di decodifica, il parametro stocastico che controlla l'appiattimento della distribuzione di probabilità sui token, non è statica, ma segue uno schedule progressivo codificato in `shared/config.py`:

```
# shared/config.py
LLM_TEMPERATURE_SCHEDULE = [0.1, 0.2, 0.4]
COMPILE_ERROR_TEMP_DOWNSHIFT = 0.1
```

La scelta risponde a una logica di esplorazione calibrata: al primo tentativo `0.1` impone un regime quasi deterministico (per bug ben diagnosticati dalla Fase 1 la soluzione più ovvia è spesso quella corretta); dopo un fallimento `0.2` introduce una diversità semantica marginale per esplorare approcci alternativi restando conservativo; nel tentativo finale `0.4` abilita una diversificazione marcata, utile a sfuggire ai *local minima* in cui il modello ripropone varianti dello stesso codice errato.

Il parametro `COMPILE_ERROR_TEMP_DOWNSHIFT` aggiunge un'ottimizzazione reattiva: se il tentativo precedente è fallito a livello sintattico (`COMPILE`) anziché semantico (`TEST`), il modello si è allontanato dalla grammatica valida, e il sistema decurta preventivamente la temperatura per forzare la convergenza verso costrutti più sicuri.

2. Isolamento della Sorgente per Ogni Tentativo

Per impedire che l'inquinamento del codice causi falsi negativi, prima di ogni tentativo il sistema (`_create_patched_copy()`) ne isola fisicamente il workspace: `shutil.copytree()` realizza una copia profonda del progetto target, così che ogni iterazione parta dal file difettoso originale, senza che patch fallite di tentativi precedenti contaminino le successive. Conservare lo stato in directory separate (`attempt_1`, `attempt_2`, ecc.) fornisce inoltre un artefatto utile all'ispezione *post-hoc*.

3. Iniezione degli Errori Precedenti

Il terzo pilastro è la memoria a breve termine che abilita il meta-apprendimento dell'agente, delegata a un modulo dedicato:

La Memoria Conversazionale: `conversation_store.py`

I sistemi APR multi-tentativo di prima generazione non ritenevano lo storico tra iterazioni: ogni tentativo ripartiva dal prompt iniziale, senza comunicare quali patch fossero già state tentate né perché fossero fallite. Questo approccio "smemorato" produceva un *loop sterile*, con il modello che rigenerava la stessa patch consumando il budget senza progressi. Per scardinarlo, `HybridRepair` estende il paradigma di *ChatRepair* [46]: la storia conversazionale di ogni tentativo fallito è persistita e distillata in un riassunto iniettato come contesto negativo nel tentativo seguente. Il modulo `patch_agent/conversation_store.py` funge da database locale per questa memoria.

Per massimizzare la resilienza I/O, `save()` storicizza la conversazione in formato JSONL (*JSON Lines*): poiché ogni messaggio (ruolo, contenuto e chiamate agli strumenti) è un record indipendente su una riga, un'interruzione del processo non corrompe i record già scritti, a differenza di un array JSON monolitico in cui la parentesi di chiusura mancante invaliderebbe l'intero file.

Quando l'agente sta per iniziare il tentativo $k+1$, il sistema interroga `conversation_store.py` per costruire il ponte cognitivo tra fallimento passato e nuovo sforzo, tramite `get_failure_summary()`:

```
def get_failure_summary(self) -> str:
    """
    Costruisce un riassunto dei tentativi falliti passati
    utilizzando il costrutto di negative prompting.
    """
    failed_attempts = self._load_previous_failures()
    if not failed_attempts:
        return ""

    summary = "DO NOT REPEAT THESE APPROACHES:\n\n"
    for attempt in failed_attempts:
        summary += f"Attempt {attempt.id} Action:\n"
        summary += f"{attempt.last_assistant_message}\n"
        summary += f"Resulting Error:\n{attempt.latest_error}\n
           \n"

    return summary
```

Invece di chiedere genericamente di "cercare un'altra via", la stringa esplicita quale azione ha compiuto l'assistente nell'ultimo messaggio del tentativo fallito e quale errore ha scatenato. La testata `"DO NOT REPEAT THESE APPROACHES"` agisce da *negative prompting* forte: fornire esempi espliciti con un imperativo negativo penalizza le sequenze simili al testo negato, spingendo l'agente verso ragionamenti inesplorati e rendendo le iterazioni effettivamente progressive.

Il Meccanismo di Retry all'Opera: i Due Tentativi di Csv-4

La riparazione del bug `Csv-4`, usato come esempio guida lungo la trattazione, richiede esattamente due tentativi e mostra i tre pilastri del retry convergere su un'unica patch corretta.

Tentativo 1 (temperatura 0.1).

Partendo dalla diagnosi della Fase 1 (`NullPointerException` in `getHeaderMap()`, riga 288), il modello modifica *due* metodi: fa restituire a `getHeaderMap()` direttamente `this.headerMap` e inizializza `hdrMap = new LinkedHashMap<>()` in `initializeHeader()`. Il codice **compila**, superando la fase `COMPILE`, ma cede in fase `TEST`: rimuovendo l'origine del `null`, `getHeaderMap()` ora restituisce una mappa vuota dove il contratto del metodo prevede `null`.

```
java.lang.AssertionError: expected null, but was:<{}>
    at org.apache.commons.csv.CSVParserTest.testNoHeaderMap(
       CSVParserTest.java:670)
```

L'esito è `TEST_FAIL` (47 test su 48 superati): la FSM transita nello stato `RETRY`.

Iniezione del fallimento.

Prima di avviare il secondo tentativo, `get_failure_summary()` distilla l'azione fallita e l'errore prodotto in un contesto negativo, iniettato nel prompt successivo:

```
DO NOT REPEAT THESE APPROACHES:
Attempt 1 Action: return this.headerMap;  (+ hdrMap = new
   LinkedHashMap<>())
Resulting Error: expected null, but was:<{}>
```

Tentativo 2 (temperatura 0.2).

Guidato dal feedback e operando su una copia pulita del sorgente (sandbox `attempt_2`), l'agente abbandona la doppia modifica e converge su un fix minimale e rispettoso del contratto, circoscritto al solo `getHeaderMap()`:

```
return this.headerMap == null ? null : new LinkedHashMap<>(this
   .headerMap);
```

Questa volta l'esito è `PASS` (48 test su 48) e la validazione semantica conferma il rispetto del contratto documentato. I tre pilastri del retry, temperatura crescente ($0.1 \rightarrow 0.2$), isolamento del workspace per tentativo e memoria del fallimento, convergono così in una singola riparazione corretta, chiudendo il ciclo della FSM nello stato `DONE`.

3.6 Fase 3 - PatchAgent: loop di tool-use e toolkit

3.6.1 Il Loop di Tool-Use: `AzureClient.chat_with_tools`

L'implementazione concreta e operativa del ciclo iterativo dell'agente risiede all'interno del modulo `services/azure_client.py`, ed è orchestrata dal metodo `chat_with_tools()`. Questo componente è responsabile della gestione a basso livello dell'interazione tra il *Large Language Model* e il toolkit di strumenti a sua disposizione.

L'Architettura Function-Calling dell'API OpenAI

Il fondamento è l'architettura *Function Calling* dell'API *Chat Completions* di OpenAI, un protocollo per la comunicazione bidirezionale tra modello e codice applicativo. Quando l'esecuzione di uno strumento è necessaria, l'API restituisce un oggetto `tool_calls` anziché un messaggio di testo: una lista di invocazioni, ciascuna con identificatore univoco (`tool_call_id`), nome della funzione (`function.name`) e argomenti serializzati in JSON (`function.arguments`). Il codice applicativo intercetta la richiesta, esegue lo strumento Python e inietta i risultati nella conversazione come messaggi con ruolo `role: "tool"` e il corretto `tool_call_id`. Il metodo `chat_with_tools()` astrae questo protocollo, come illustrato di seguito:

```
def chat_with_tools(self, messages: list, tools: list,
    tool_executor, max_tool_rounds: int) -> tuple:
    """Gestisce il loop conversazionale con invocazione
       automatica dei tool."""
    for _ in range(max_tool_rounds):
        # Chiamata all'API con retry esponenziale
        response = self._call_with_retry(messages, tools)
        response_message = response.choices[0].message

        # Mutazione in-place della storia conversazionale
        messages.append(response_message)

        # Se non ci sono tool calls, il modello ha generato il
           testo finale
        if not response_message.tool_calls:
            return response_message.content, messages

        # Esecuzione sequenziale delle funzioni richieste
        for tool_call in response_message.tool_calls:
            func_name = tool_call.function.name
            try:
                # Deserializzazione degli argomenti generati
                   dal modello
                args = json.loads(tool_call.function.arguments)
            except json.JSONDecodeError:
                # Fallback di sicurezza in caso di
                   allucinazione sintattica
                args = {}

            # Esecuzione fisica dello strumento (dispatch all'
               executor iniettato)
```

```
            result = tool_executor(func_name, args)

            # Mutazione in-place: aggiunta del risultato
            messages.append({
                "role": "tool",
                "tool_call_id": tool_call.id,
                "content": str(result)
            })

    # Raggiungimento del limite massimo di round
    return "", messages
```

Due aspetti implementativi meritano attenzione:

La mutazione *in-place* della lista `messages`: la lista è passata per riferimento e modificata con `.append()` a ogni iterazione, così che il chiamante (`PatchAgent._run_single_attempt()`) abbia sempre l'intera storia conversazionale sincronizzata, anche in caso di terminazione prematura. L'oggetto `updated_messages` restituito non è una copia, ma la stessa istanza mutata.

Il blocco try/except per `json.loads()`: gli argomenti generati dal modello vanno deserializzati da JSON, ma l'LLM può produrre JSON malformato. Il blocco `try/except json.JSONDecodeError` intercetta l'anomalia e applica un fallback a dizionario vuoto (`args = {}`): lo strumento fallirà la validazione interna restituendo un errore testuale che, nel round successivo, l'LLM userà per correggere autonomamente l'invocazione. Soluzione architetturalmente superiore al crash.

Il Budget di Tool-Call e la Prevenzione di Loop Infiniti

Un agente libero di invocare strumenti in un ciclo `while` rischia di innescare loop infiniti, esaurendo il budget dell'API senza convergere. Il parametro `max_tool_rounds` agisce da *safety limit*, derivato come `config.MAX_AGENT_STEPS * 3`. Con `MAX_AGENT_STEPS = 3`, l'agente dispone di 9 round. Il moltiplicatore riflette l'anatomia della FSM: ciascuna macro-fase (`EXPLORE`, `PATCH`, `COMPILE`, `TEST`) richiede circa un round, ma la fase `COMPILE` può innescare cicli di correzione inline. Moltiplicare il limite base fornisce un margine per assorbire un intero ciclo FSM con un paio di riparazioni sintattiche al volo, mantenendo un tetto rigido contro i loop viziosi.

Il Retry Esponenziale per Errori Transitori

L'affidabilità di un sistema cloud dipende dalla resilienza agli errori di rete e al rate-limiting imposto dalle API di Azure OpenAI. L'interazione è delegata al wrapper `_call_with_retry()`, che implementa un backoff esponenziale:

```
def _call_with_retry(self, messages: list, tools: list):
    """Esegue la chiamata API gestendo gli errori transitori (
       rate-limit, timeout, connessione)."""
    delays = [2, 4, 8]  # Backoff esponenziale in secondi

    for attempt_idx, delay in enumerate(delays):
        try:
            return self.client.chat.completions.create(
```

```
                model=self.model,
                messages=messages,
                tools=tools
            )
        except (openai.RateLimitError, openai.APITimeoutError,
           openai.APIConnectionError) as e:
            if attempt_idx == len(delays) - 1:
                # Esauriti i retry, propaga l'eccezione
                raise e
            # Attende prima di ritentare
            time.sleep(delay)
```

Il wrapper ritenta sui principali errori transitori (rate-limit, timeout e interruzioni di connessione); l'intervallo di attesa crescente (2, 4, 8 secondi) è il protocollo raccomandato per il `RateLimitError`, ed è particolarmente vitale in modalità batch (es. `run_all_bugs.py`). Quando decine di thread paralleli vengono respinti per rate-limit, un'attesa fissa li farebbe ripresentare in blocco contro i server (*thundering herd*), causando un nuovo collasso. Il backoff esponenziale scagliona le ritrasmissioni nel tempo, disperdendo i picchi di carico e riducendo la probabilità di reimpattare contro i limiti.

3.6.2 Il Toolkit dell'Agente: Sei Strumenti per la Riparazione Autonoma

Il modulo `patch_agent/tools.py` definisce l'arsenale operativo del `PatchAgent`: sei strumenti esposti al modello tramite gli schemi JSON del *Function Calling* e implementati come funzioni Python. Lo stato è gestito con il pattern *Context Object*: anziché passare singolarmente percorsi e classi di test a ogni funzione, il contesto del tentativo corrente è aggregato in un oggetto mutabile condiviso:

```
@dataclass
class ToolContext:
    bug_dir: Path
    patched_dir: Path
    test_classes: list[str]
```

L'iniezione del `ToolContext` riduce l'accoppiamento tra il dispatcher (`create_tool_executor()`) e le singole implementazioni, rendendo il toolkit estensibile: aggiungere strumenti non richiede modifiche alla firma di routing.

`read_file`: Lettura del Codice Sorgente con Numerazione di Riga

Il primo strumento è `read_file`, per ispezionare i file del progetto durante la fase esplorativa.

```
def tool_read_file(context: ToolContext, file_path: str) -> str
    :
    """Legge il contenuto di un file sorgente applicando
       numerazione di riga."""
    target_path = context.patched_dir / file_path

    with open(target_path, 'r', encoding='utf-8') as f:
        content = f.read()
```

```
# Troncamento per preservare il budget di token
if len(content) > 30000:
    content = content[:30000] + "\n... [TRUNCATED]"

lines = content.split('\n')
# Numerazione destra-allineata a 4 cifre
numbered_lines = [f"{i+1:4d} | {line}" for i, line in
   enumerate(lines)]
return '\n'.join(numbered_lines)
```

Due dettagli sono cruciali. La formattazione `f"{i+1:4d} | {line}"` produce una numerazione di riga allineata a 4 cifre: GPT-4, pre-addestrato su diff di GitHub con formattazione analoga, riesce così a riferirsi a una riga per numero e a fornirla come parametro `target_line` al patching. Il limite di 30.000 caratteri evita che file Java monolitici (come le classi di JFreeChart) saturino il *token budget*, costringendo l'agente a una navigazione più mirata.

`search_symbol`: Ricerca Testuale nel Source Tree

Quando il file in esame non contiene le risposte necessarie, l'agente può esplorare l'intero albero dei sorgenti tramite `search_symbol`. Lo strumento è un wrapper attorno a `grep -rnI --include=*.java`, che esegue una ricerca *case-sensitive* ricorsiva sui soli file Java; il flag `-I` ignora i file binari (es. `.class`) che genererebbero output illeggibile. In fase `EXPLORE` serve a localizzare l'implementazione di un metodo citato nello stack trace, trovare le assegnazioni di una variabile `null` o individuare implementazioni multiple di un'interfaccia. L'output è troncato a 20 risultati.

`get_method_signature`: Verifica delle API Reali

Per risolvere le ambiguità del sistema di tipi di Java, l'agente dispone di `get_method_signature`, che ispeziona dinamicamente le firme dei metodi. Lo strumento riutilizza la funzione `_extract_public_methods()` di `ingredient_forge.py` (Fase 2): condividere l'estrattore garantisce che le API esplorate dinamicamente abbiano la stessa struttura testuale di quelle fornite negli "Ingredients". Il parametro `method_name` filtra i risultati per ridurre il rumore delle classi con molti metodi.

`apply_ast_patch`: L'Atto di Riparazione Strutturale

Questo strumento sancisce la transizione dallo stato esplorativo allo stato esecutivo (`PATCH`), materializzando la modifica al codice. La funzione delega il lavoro pesante a `apply_and_write()` del modulo `ast_applier.py`, cuore della manipolazione AST/CST via Tree-sitter. Il diff unificato restituito al modello ha funzione puramente diagnostica: offre una conferma visiva di ciò che è stato alterato, così che l'agente possa accorgersi di eventuali rimozioni accidentali di righe ancor prima di compilare.

`compile_check` e `run_tests`: Gli Oracoli di Validazione

Gli ultimi due strumenti chiudono il ciclo agendo da oracoli di verifica: `compile_check` controlla che il codice modificato sia sintatticamente valido (fase `COMPILE`), `run_tests`

che superi la suite di test (fase `TEST`). Per ragioni di isolamento e tolleranza ai guasti (timeout, loop infiniti, dipendenze mancanti), nessuno dei due lancia comandi shell direttamente: entrambi delegano l'esecuzione al sandbox di `sandbox_evaluator.py`.

`compile_check`: compila la sorgente tramite `compile_patched_source()`; se la compilazione fallisce, restituisce l'output di `javac` troncato a 3000 caratteri, in modo da mostrare all'agente l'errore primario senza sommergerlo nella cascata di errori secondari che ne derivano.

`run_tests`: riesegue la suite JUnit tramite `evaluate()` e restituisce uno fra tre esiti, `PASS`, `COMPILE_ERROR` oppure `TEST_FAIL` (quest'ultimo corredato dello stack trace), che dicono all'agente se la patch ha funzionato e, in caso negativo, dove ha ceduto.

3.7 Fase 3 - PatchAgent: riparazione strutturale e validazione

3.7.1 Riparazione Strutturale via Tree-sitter: CST vs AST

L'approccio alla modifica del codice sorgente è uno degli snodi più delicati nella progettazione di un sistema APR [17]: tradurre un'intenzione correttiva in una mutazione fisica richiede un meccanismo che garantisca accuratezza della modifica e validità formale dell'artefatto. `HybridRepair` adotta la manipolazione diretta del *Concrete Syntax Tree* tramite Tree-sitter, distaccandosi dai paradigmi testuali classici.

I Limiti Fondamentali degli Approcci Testuali

Prima di esplorare le specificità tecniche di Tree-sitter, è imperativo analizzare criticamente le ragioni per cui gli approcci tradizionali, manipolazione tramite espressioni regolari (regex), diff unificati e operatori di mutazione, risultano inadeguati per un sistema APR di alta affidabilità.

Il fallimento delle espressioni regolari e il problema del bilanciamento
Le espressioni regolari operano su flussi monodimensionali di caratteri, in contrasto con la natura gerarchica e ricorsiva del codice Java: un metodo non è una sequenza lineare ma una struttura annidata (`if/else`, cicli, `try/catch`, classi anonime, lambda), in cui ogni livello introduce parentesi graffe da bilanciare. Per la teoria dei linguaggi formali, il bilanciamento delle parentesi appartiene ai linguaggi *context-free*, mentre le regex descrivono solo linguaggi regolari [2]: una regex semplice cattura solo metodi piatti, fallendo al primo blocco annidato. La prima versione di `HybridRepair` tentava di arginare il problema combinando regex e contatori manuali di graffe (`_count_braces()`), ma ciò scatenava il "problema P1": patch con parentesi sbilanciate o dichiarazioni frammentate, con un alto tasso di scarto.

Il fallimento dei diff unificati nei sistemi multi-tentativo Il formato *unidiff* (usato da `git diff` e da vecchi sistemi APR) codifica una modifica come un *hunk* ancorato a righe di contesto preesistenti, assumendone l'immutabilità. In un agente iterativo guidato da un LLM questa assunzione crolla: i diff generati fanno spesso riferimento al sorgente originale del prompt, ma al procedere dei round il file bersaglio accumula variazioni o stati transitori. Il conseguente disallineamento posizionale causa il rigetto del diff in applicazione, generando un falso negativo: la logica della patch può essere corretta, ma il suo veicolo testuale ne impedisce la materializzazione [19, 46].

Il fallimento dei mutanti e l'overfitting L'ultimo approccio tradizionale, basato sui mutanti, altera l'AST con schemi predefiniti (operatori di mutazione: forzare una condizione a `true`, sostituire un operatore relazionale, inserire null-check). Soffre di un doppio limite: lo spazio esplorativo è confinato agli operatori implementati (se il fix richiede una logica non codificata, la riparazione è impossibile) e, soprattutto, i mutanti inducono un massiccio *overfitting*, generando patch che bypassano il test fallito in modo accidentale e degradano comportamenti per input non coperti. Il tasso di overfitting può superare il 90% delle patch plausibili [32, 17].

Introduzione a Tree-sitter: il Parser Incrementale ad Alta Fedeltà

Per superare queste fragilità, `HybridRepair` adotta Tree-sitter, una libreria di parsing incrementale open-source sviluppata da GitHub. A differenza dei parser dei compilatori Java (ECJ, `javac`), Tree-sitter è guidato da una grammatica GLR (*Generalized Left-to-Right Rightmost*). Il suo vantaggio è la resilienza: mentre i parser LL(k)/LR(k) interrompono l'analisi al primo errore sintattico distruggendo l'albero parziale, Tree-sitter incapsula le anomalie in nodi `ERROR` e completa comunque il parsing dell'intero documento, capacità critica nell'APR, dove il sorgente è per definizione difettoso. L'integrazione della libreria nativa C nell'ecosistema Python è gestita con un pattern *lazy singleton*, dettagliato nella Sezione 3.7.2.

La Distinzione Cruciale: AST del Compilatore vs CST di Tree-sitter

Un errore comune è confondere AST e CST. Il salto qualitativo di Tree-sitter sta nella dicotomia tra astrazione semantica (AST) e fedeltà lessicale (CST).

L'Astrazione Semantica dell'AST del Compilatore Un compilatore come `javac` mira a produrre bytecode, perciò il suo AST (*Abstract Syntax Tree*) scarta tutto ciò che non impatta l'esecuzione: commenti, spazi bianchi, indentazione, parentesi ridondanti, punti e virgola. Questa astrazione è preziosa per la compilazione ma distruttiva per il patching: modificare un nodo e poi rigenerare il sorgente (*pretty-printing*) ricostruirebbe l'intero file, disgregando indentazione e stile originari e riversando modifiche estetiche su righe non pertinenti, rendendo impossibile validare il diff.

La Fedeltà Lessicale del CST di Tree-sitter Tree-sitter, al contrario, produce un Concrete Syntax Tree (CST) che non scarta nulla: ogni commento, spazio, ritorno a capo, annotazione e parentesi è un nodo fisico. Il pregio decisivo è il posizionamento: ogni nodo trasporta uno `start_byte` e un `end_byte`, offset precisi al byte nel file originale. Il frammento seguente illustra il CST di una dichiarazione di metodo Java:

```
method_declaration [byte: 0..46]
  modifiers: modifiers [byte: 0..6]
    public [byte: 0..6]
  type: void_type [byte: 7..11]
  name: identifier [byte: 12..16] "test"
  parameters: formal_parameters [byte: 16..18]
    ( [byte: 16..17]
    ) [byte: 17..18]
  body: block [byte: 19..46]
    { [byte: 19..20]
      ...
    } [byte: 45..46]
```

Grazie agli indici di ancoraggio (es. `[byte: 0..46]`), `HybridRepair` estrae, rimpiazza o inietta blocchi operando in aritmetica sui byte, con la certezza che lo slice `source[node.start_byte:node.end_byte]` riproduce fedelmente il testo originale, tabulazioni e punteggiatura incluse.

Il Vocabolario dei Nodi Tree-sitter per Java

La grammatica di `tree-sitter-java` conta oltre 100 tipi di nodi. `HybridRepair` usa prevalentemente quelli che forniscono i confini delle macro-strutture del codice, riassunti nella Tabella 3.3.

Tabella 3.3: Principali tipi di nodo della grammatica `tree-sitter-java` usati da `HybridRepair`.

Tipo nodo	Descrizione	Campo figlio rilevante
`method_declaration`	Dichiarazione di metodo	`name` (`identifier`), `body` (`block`)
`constructor_declaration`	Dichiarazione di costruttore	`name` (`identifier`), `body` (`block`)
`field_declaration`	Campo di classe	`type`, `declarator`
`local_variable_declaration`	Variabile locale	`type`, `declarator`
`formal_parameter`	Parametro formale di metodo	`type`, `name`
`class_declaration`	Dichiarazione di classe	`name`, `body`
`block`	Blocco di istruzioni	child nodes
`identifier`	Identificatore testuale Java	(testo del nodo)

A partire da questo vocabolario, la localizzazione del nodo del metodo target nel CST è affidata a una visita *Depth-First Search* sui soli nodi `method_declaration` e `constructor_declaration`: il confronto avviene sul campo `name` (l'`identifier` del metodo) e restituisce il nodo portatore dei *byte offset* su cui agire. L'algoritmo completo, comprensivo della disambiguazione degli overload, è dettagliato nella Sezione 3.7.2.

3.7.2 Precisione Chirurgica e Modifiche Posizionali

Il modulo `patch_agent/ast_applier.py` implementa il meccanismo centrale della riparazione strutturale. La sua funzione principale, `apply_method_replacement()`, concretizza la sostituzione posizionalmente precisa di un metodo, in tre passaggi: parsing via Tree-sitter, localizzazione del nodo del metodo target e sostituzione del relativo span di testo.

L'Inizializzazione Lazy del Parser: `_ensure_tree_sitter()`

Il primo passo è l'istanza del parser, gestita con il pattern *lazy singleton*:

```
def _ensure_tree_sitter():
    """Inizializzazione lazy del parser Tree-sitter come
       singleton."""
    global _TS_AVAILABLE, _JAVA_LANGUAGE, _PARSER

    if not _TS_AVAILABLE:
        from tree_sitter import Language, Parser
        import tree_sitter_java

        # Caricamento della grammatica compilata nativa
        _JAVA_LANGUAGE = Language(tree_sitter_java.language(),
            'java')
        _PARSER = Parser()
        _PARSER.set_language(_JAVA_LANGUAGE)
```

```
            _TS_AVAILABLE = True

    return _PARSER
```

Il pattern interviene su un collo di bottiglia: l'inizializzazione di Tree-sitter (caricamento della grammatica pre-compilata via binding C) costa decine di millisecondi, mentre il parsing di un file ne richiede pochi. Ritardando l'istanza al primo utilizzo e condividendo l'oggetto globale `_PARSER`, il sistema evita ritardi latenti durante l'esecuzione batch. La funzione è inoltre condivisa tra `ast_applier.py` (riparazione) e `prolog_grounding.py` (risoluzione dei tipi nel *grounding*): grazie al sistema di import di Python, le variabili globali del modulo sono caricate una sola volta per processo, garantendo un'istanza unica tra tutti gli importatori.

La Funzione di Parsing: `_parse_java()`

Ottenuto il parser, il sorgente è analizzato dalla funzione `_parse_java()` per generare il CST. Il dettaglio critico è la conversione esplicita della stringa Python in byte UTF-8 (`source_code.encode('utf-8')`) prima di `parser.parse()`: Tree-sitter opera nello spazio dei byte, e gli offset `start_byte`/`end_byte` sono offset in byte, non in caratteri Unicode. Per file ASCII i due coincidono, ma in presenza di stringhe o commenti non-ASCII (frequenti nei progetti internazionali di Defects4J) l'uso di character offset provocherebbe disallineamenti silenti che corromperebbero la patch. Operare sui byte UTF-8 è quindi essenziale per la precisione.

La Strategia di Sostituzione via AST: `_replace_via_ast()`

Individuate le coordinate esatte del metodo nell'albero, la sostituzione fisica è eseguita da `_replace_via_ast()`:

```
def _replace_via_ast(source_bytes: bytes, method_node,
    new_body_indented: str) -> bytes:
    """Esegue la sostituzione chirurgica sfruttando gli offset
       di Tree-sitter."""

    # Estrazione degli offset posizionali dal nodo
    start_byte = method_node.start_byte
    end_byte = method_node.end_byte

    # Codifica della patch generata dall'LLM
    new_body_bytes = new_body_indented.encode('utf-8')

    # Slicing posizionale e concatenazione
    patched_bytes = source_bytes[:start_byte] + new_body_bytes
       + source_bytes[end_byte:]

    return patched_bytes
```

La robustezza risiede nell'operazione di slicing e concatenazione `source_bytes[:start_byte] + new_body_bytes + source_bytes[end_byte:]`: la prima porzione conserva tutto ciò che precede il metodo (package, import, classe, metodi precedenti), `new_body_bytes` è la nuova implementazione re-indentata e codificata, e la terza recupera tutto ciò che segue (graffa di chiusura, metodi successivi).

Poiché gli attributi `start_byte`/`end_byte` delineano in modo esatto l'intervallo del nodo, ne deriva una correttezza sintattica per costruzione: se il file originale è valido, `new_body_indented` è un metodo valido con il medesimo identificatore e `method_node` ne inquadra le coordinate, allora `patched_bytes` è garantito sintatticamente integro. A differenza degli approcci testuali, la correttezza non è una scommessa statistica ma una conseguenza della manipolazione posizionale. Eventuali errori residui saranno intercettati dal compilatore nella fase `COMPILE`.

Il Rilevamento del Metodo Target: `_find_method_by_name()`

Per alimentare la sostituzione col nodo corretto, il sistema deve prima localizzarlo nel CST tramite `_find_method_by_name()`, che gestisce gerarchie profonde e ambiguità:

```
def _find_method_by_name(root_node, target_name: str,
    target_line: int = None):
    """Ricerca iterativa DFS del metodo target con
       disambiguazione degli overload."""
    stack = [root_node]
    candidate_nodes = []

    while stack:
        node = stack.pop()

        # Identificazione di metodi o costruttori
        if node.type in ['method_declaration', '
           constructor_declaration']:
            name_node = node.child_by_field_name('name')
            if name_node and name_node.text.decode('utf-8') ==
               target_name:
                candidate_nodes.append(node)

        # Inserimento inverso dei figli per preservare la
           visita left-to-right
        for child in reversed(node.children):
            stack.append(child)

    # Disambiguazione in presenza di overload multipli
    if target_line is not None and len(candidate_nodes) > 1:
        for node in candidate_nodes:
            # Tree-sitter restituisce righe zero-indexed,
               aggiungiamo 1
            start_row = node.start_point[0] + 1
            end_row = node.end_point[0] + 1

            # Verifica se la linea difettosa ricade nei confini
               del metodo
            if start_row <= target_line <= end_row:
                return node

    return candidate_nodes if candidate_nodes else None
```

Due aspetti spiccano. L'esplorazione del CST usa una *Depth-First Search* iterativa con stack esplicito anziché ricorsiva: nei file Java enterprise l'annidamento può raggiungere profondità estreme (classi anonime, `try/catch`, iterazioni annidate) e un DFS ricorsivo rischierebbe di superare il limite dello stack di chiamate Python (1000 frame) con un `RecursionError`; lo stack iterativo sull'heap immunizza l'algoritmo.

In secondo luogo, la funzione disambigua gli overload tramite il parametro `target_line`. In Defects4J, classi come `DatasetUtilities` (JFreeChart) o `StringUtils` (Apache Commons) presentano spesso dozzine di metodi omonimi con firme diverse: cercare per solo nome (più nodi in `candidate_nodes`) applicherebbe la patch al primo a caso. Ricevendo la `target_line` (propagata dal `PatchAgent` a partire da `FaultLocation`), l'algoritmo confronta la coordinata con il *bounding box* di ogni candidato (`start_point`/`end_point`) e seleziona solo il nodo che abbraccia la riga difettosa, risolvendo l'ambiguità.

La Preservazione dell'Indentazione: `_detect_indentation()` e `_reindent()`

La preservazione dello stile di indentazione originale è un requisito critico: iniettare un metodo con indentazione discorde produrrebbe un diff "inquinato", in cui ogni riga risulta modificata per discrepanze di spazi bianchi anziché solo quelle realmente alterate, rendendo la patch difficile da revisionare e aumentando il rischio di errori legati al whitespace in pipeline CI con regole di stile stringenti (Checkstyle). Il sistema usa due funzioni dedicate: `_detect_indentation()` per campionare lo stile originario e `_reindent()` per applicarlo al codice generato.

La funzione `_detect_indentation()` estrae il prefisso (spazi o tabulazioni) della prima riga del metodo originale. Nel dataset Defects4J le convenzioni variano molto (JFreeChart usa 4 spazi, alcuni sottoprogetti Apache Commons le tabulazioni, Closure Compiler 2 spazi): il rilevamento dinamico svincola il sistema dal conoscere a priori le policy del progetto.

Una volta estratto il prefisso, il codice sostitutivo fornito dall'agente deve essere riallineato tramite `_reindent()`:

```
def _reindent(new_body: str, target_indent: str) -> str:
    """Applica una trasformazione di indentazione relativa al
       nuovo corpo del metodo."""
    lines = new_body.split('\n')
    if not lines:
        return new_body

    # Calcolo dell'indentazione base (livello esterno) del
       codice LLM
    base_indent_len = len(lines[0]) - len(lines[0].lstrip())
    base_indent = lines[0][:base_indent_len]

    reindented_lines = []
    for line in lines:
        # Gestione esplicita delle righe vuote
        if not line.strip():
            reindented_lines.append("")
            continue
```

```
            # Trasformazione relativa: sostituisce l'indentazione
               base con quella target
            if line.startswith(base_indent):
                reindented_lines.append(target_indent + line[
                   base_indent_len:])
            else:
                reindented_lines.append(target_indent + line.lstrip
                   ())

    return '\n'.join(reindented_lines)
```

L'algoritmo applica un'indentazione *relativa* anziché assoluta: un approccio assoluto appiattirebbe ogni riga a un margine fisso, distruggendo la struttura visiva. Qui si calcola il delta tra l'indentazione esterna del codice generato dall'LLM e il `target_indent` del file, applicandolo uniformemente a ogni riga: così l'indentazione interna del metodo (annidamenti di `if`, cicli, `try/catch`) è preservata e l'intero blocco è traslato al livello corretto. La gestione esplicita delle righe vuote (`if not line.strip()`) preserva le spaziature verticali di separazione logica.

Il Calcolo del Diff Unificato

Sebbene il sistema abbia definitivamente abbandonato i diff come veicolo per applicare la modifica, il formato *unidiff* mantiene un ruolo di primaria importanza a livello diagnostico. La funzione `_compute_diff()` (un wrapper su `difflib.unified_diff`) è invocata solo *dopo* l'applicazione via Tree-sitter e ha valore puramente diagnostico: il risultato non altera mai il file, ma è restituito all'agente come feedback di conferma (`"[SUCCESS] Patch applied ... Diff: ..."`) per validare visivamente l'impatto. Le righe di contesto sono fissate a `n=3` come nella convenzione di `git diff`: valori superiori sarebbero più leggibili ma consumerebbero inutilmente il *token budget*.

La Strategia di Fallback Graduale

L'orchestrazione finale è nel metodo `apply_method_replacement()`, che integra parsing, indentazione, manipolazione dell'AST e generazione del diff in un'unica interfaccia. Per la massima resilienza, implementa una strategia di fallback graduale a due livelli:

```
def apply_method_replacement(file_path: Path, method_name: str,
    new_body: str, target_line: int = None) -> tuple[bool, str]:
    """Applica la sostituzione del metodo tramite una strategia
        di fallback graduale."""
    original_source = file_path.read_text(encoding='utf-8')

    # 1. Preservazione stile
    target_indent = _detect_indentation(original_source)
    new_body_indented = _reindent(new_body, target_indent)

    # 2. LIVELLO PRIMARIO: Manipolazione AST via Tree-sitter
    try:
        tree, source_bytes = _parse_java(original_source)
        method_node = _find_method_by_name(tree.root_node,
           method_name, target_line)

        if method_node:
            patched_bytes = _replace_via_ast(source_bytes,
               method_node, new_body_indented)
            patched_source = patched_bytes.decode('utf-8')

            file_path.write_text(patched_source, encoding='utf
               -8')
            diff = _compute_diff(original_source,
               patched_source, file_path.name)
            return True, f"[SUCCESS] Patch applied successfully
               .\nDiff:\n{diff}"
    except Exception as e:
        # Errore non fatale, si procede al livello di fallback
        pass

    # 3. LIVELLO SECONDARIO: Fallback Regex
    try:
        # L'approccio regex richiede obbligatoriamente il
           target_line
        patched_source = _apply_regex_fallback(original_source,
            method_name, new_body_indented, target_line)

        if patched_source:
            file_path.write_text(patched_source, encoding='utf
               -8')
            diff = _compute_diff(original_source,
               patched_source, file_path.name)
            return True, f"[WARNING] Patch applied via Regex
               fallback.\nDiff:\n{diff}"
```

```
    except Exception as e:
        pass

    # 4. LIVELLO FINALE: Fallimento Graceful
    return False, "[FAILED] Failed to apply patch: method
       boundaries not found or parsing failed."
```

La ridondanza definisce un ordine di priorità decrescente in affidabilità. Il Livello Primario tenta la sostituzione via Tree-sitter AST, che garantisce correttezza sintattica per costruzione, gestione degli overload e compatibilità con annotazioni e annidamenti arbitrari. Se il parsing fallisce (raro, ma possibile con sorgente severamente corrotto) o la libreria nativa `tree-sitter-java` non è accessibile, subentra il Livello Secondario, basato su regex: opera solo su file Java standard, richiede il parametro `target_line` e, pur rischiando di sbilanciare parentesi in metodi complessi, offre una via di fuga in ambienti privi di dipendenze binarie. Se anche l'estrazione testuale fallisce, interviene il *graceful failure*: anziché sollevare un'eccezione che farebbe crashare l'agente, il sistema restituisce una tupla di errore testuale iniettata nel contesto dell'LLM, che nel round successivo può tentare un approccio diverso. Questa *graceful degradation* è imprescindibile per un sistema APR che processa batch massivi di bug eterogenei.

La Riparazione Strutturale all'Opera: Csv-4

Concretizzando il meccanismo su Csv-4: quando l'agente emette la tool-call `apply_ast_patch` per `getHeaderMap` (cfr. Sezione 3.5), `apply_method_replacement()` entra nel livello primario. `_find_method_by_name()` percorre il CST e individua l'unico nodo `method_declaration` di nome `getHeaderMap` (senza overload, la `target_line` non serve); `_detect_indentation()` rileva i 4 spazi dello stile di `CSVParser.java` e `_reindent()` vi allinea il nuovo corpo; infine `_replace_via_ast()` sostituisce lo slice `source[node.start_byte:node.end_byte]` con il metodo re-indentato. Il `_compute_diff()` restituito all'agente è un *single-hunk* pulito:

```
--- a/org/apache/commons/csv/CSVParser.java
+++ b/org/apache/commons/csv/CSVParser.java
@@ -285,7 +285,7 @@
     public Map<String, Integer> getHeaderMap() {
-        return new LinkedHashMap<String, Integer>(this.
   headerMap);
+        return this.headerMap == null ? null : new
   LinkedHashMap<>(this.headerMap);
     }
```

Il fatto che il diff tocchi una sola riga, e non l'intero metodo, è la prova empirica dell'efficacia della preservazione dell'indentazione: senza il `_reindent()` relativo, ogni riga del metodo risulterebbe modificata per sole discrepanze di spazi bianchi.

3.7.3 Il Sandbox di Valutazione: `sandbox_evaluator.py`

Il modulo `services/sandbox_evaluator.py` implementa il sandbox di valutazione, l'ambiente isolato che fornisce all'agente i due oracoli deterministici della FSM: il compilatore Java (`javac`) e il runner dei test (`org.junit.runner.JUnitCore`). Il termine "sandbox" denota un isolamento operativo: ogni tentativo agisce su una copia fisica separata del codice, in una directory dedicata, così che fallimenti e mutazioni di un tentativo non inquinino i successivi.

La Gestione del Classpath: `config.properties` e Glob Expansion

Prima di compilare o eseguire il codice, l'ambiente deve risolvere le dipendenze. In Defects4J la configurazione è nel file `config.properties` nella directory radice di ogni bug, estratto dalla funzione `_read_classpath()`:

```
def _read_classpath(bug_dir: Path) -> str:
    """Estrae e risolve il classpath dal file config.properties
       ."""
    props_path = bug_dir / "config.properties"

    # config.properties non ha un'intestazione di sezione: la
       si inietta
    # per poterlo dare in pasto a configparser
    raw = "[DEFAULT]\n" + props_path.read_text(encoding="utf-8"
       )
    parser = configparser.RawConfigParser()
    parser.read_string(raw)
    cp_raw = parser.get("DEFAULT", "class.path", fallback="")

    resolved_entries = []
    for entry in cp_raw.split(":"):
        resolved = (bug_dir / entry.strip()).resolve()
        if '*' in str(resolved):
            # Glob expansion per le wildcard (es. ../../lib/*)
            resolved_entries += [str(j) for j in resolved.
               parent.glob("*.jar")]
        else:
            # Risoluzione in percorso assoluto
            resolved_entries.append(str(resolved))

    return ":".join(resolved_entries)
```

Il file definisce il classpath con percorsi relativi alla directory del bug (es. `class.path=./../lib/junit-4.11.jar:...`); poiché `config.properties` non possiede un'intestazione di sezione, si antepone `[DEFAULT]` per poterlo analizzare con `configparser`, quindi ogni voce è ancorata in percorso assoluto tramite `(bug_dir / entry).resolve()`. Aspetto chiave è il supporto alla *glob expansion*: alcuni progetti usano wildcard (es. `../../lib/*`) anziché elencare ogni JAR; il codice rileva il carattere `*` e raccoglie dinamicamente tutti i `.jar` della directory, assorbendo le differenze strutturali tra i repository di Defects4J.

La Compilazione Selettiva dei File Modificati

La verifica sintattica usa `javac`, ma ricompilare un intero progetto enterprise (JFreeChart, Closure Compiler) a ogni tentativo richiederebbe minuti. Il sandbox supera il problema con una compilazione selettiva:

```
def _find_modified_files(original_dir: Path, patched_dir: Path)
    -> list[str]:
    """Identifica i file Java modificati tramite confronto byte
       -a-byte."""
    import filecmp
    comparison = filecmp.dircmp(original_dir, patched_dir)
    return [f for f in comparison.diff_files if f.endswith('.
       java')]

def compile_patched_source(patched_dir: Path, bug_dir: Path,
   classpath: str):
    """Compila selettivamente il file modificato tramite javac.
       """
    import subprocess

    modified_files = _find_modified_files(bug_dir, patched_dir)
    if not modified_files:
        return True, ""

    target_file = patched_dir / modified_files[0]

    cmd = [
        "javac",
        "-cp", classpath,
        "-sourcepath", str(patched_dir),
        "-nowarn",
        "-encoding", "UTF-8",
        str(target_file)
    ]

    result = subprocess.run(cmd, capture_output=True, text=True
       )
    return result.returncode == 0, result.stderr
```

La funzione `_find_modified_files()` usa `filecmp.dircmp()` per un confronto byte-a-byte tra directory originale e sandbox, superiore a un controllo dei timestamp (che la copia tra filesystem potrebbe alterare causando falsi positivi): il confronto binario estrae il solo file mutato dall'LLM.

Isolato il file, si invoca il compilatore con due flag chiave. Il flag `-sourcepath` istruisce `javac` a risolvere i riferimenti a classi non specificate cercandone i sorgenti nella directory radice, permettendo di compilare la sola classe modificata risolvendo comunque i riferimenti incrociati ed evitando una ricompilazione totale. Il flag `-nowarn` sopprime gli avvisi non bloccanti del codice legacy (deprecazioni, unchecked assignments) che altrimenti inquinerebbero il prompt dell'agente, mantenendolo focalizzato sui veri errori sintattici.

L'Esecuzione JUnit e il Parsing dell'Output

L'oracolo finale è il collaudo della suite di test, innescato in totale automazione invocando `org.junit.runner.JUnitCore` in modalità *command-line* (`java -cp <classpath> org.junit.runner.JUnitCore <test_classes>`) entro un `timeout` di 300 secondi. Quest'ultimo è uno scudo contro un fenomeno tipico dell'APR: le modifiche dell'agente, specie su bug di sincronizzazione multithreading o I/O, possono generare deadlock o loop infiniti nei test. Un limite di 5 minuti previene il blocco indefinito dell'elaborazione batch.

Il runner JUnit in CLI non restituisce un log strutturato (XML), ma riversa sullo standard output un report testuale grezzo, analizzato dalla funzione `_parse_junit_output()`:

```
def _parse_junit_output(output: str, original_failing_tests:
    set):
    """Analizza l'output testuale JUnit ed estrae le metriche
       di progresso."""
    import re

    # Estrazione delle firme tramite espressione regolare
    failure_pattern = re.compile(r"\.E\s+([a-zA-Z0-9_]+)\(([a-
       zA-Z0-9_.]+)\)")

    current_failures = set()
    for match in failure_pattern.finditer(output):
        method_name, class_name = match.groups()
        current_failures.add(f"{class_name}::{method_name}")

    previously_failing = original_failing_tests
    still_failing = previously_failing.intersection(
       current_failures)

    # Metrica cruciale di parziale successo
    previously_failing_now_passing = previously_failing -
       still_failing

    return {
        "status": "PASS" if not current_failures and "OK (" in
           output else "TEST_FAIL",
        "failures": list(current_failures),
        "previously_failing_now_passing": list(
           previously_failing_now_passing)
    }
```

L'algoritmo usa una regex per estrarre dal log le firme dei test falliti (`current_failures`) e le incrocia con `original_failing_tests` (i fallimenti pre-patch dal `FaultReport`) per ricavare `previously_failing_now_passing`. Questa metrica è di grande utilità: spesso una patch non fa passare tutti i test simultaneamente, e misurare quanti test siano passati da falliti a corretti (es. 3 su 5) offre un indicatore direzionale di progresso, segnalando all'agente che la rotta è parzialmente valida anche senza successo totale.

3.7.4 Strutture Dati della Fase 3: `PatchAttempt` e `RepairResult`

Il contratto di output della Fase 3 è formalizzato in due `@dataclass` definite in `shared/models.py`, progettate non come semplici registri di esito ma come archivi permanenti:

```
from dataclasses import dataclass, field
from typing import Optional, List, Dict, Any

@dataclass
class PatchAttempt:
    """Archivio permanente di un singolo tentativo di
       riparazione."""
    id: int
    success: bool
    agent_messages: List[Dict[str, Any]] = field(
       default_factory=list)
    applied_diff: Optional[str] = None
    latest_error: Optional[str] = None
    test_results: Optional[Dict[str, Any]] = None

@dataclass
class RepairResult:
    """Artefatto finale aggregato dell'intera pipeline
       HybridRepair."""
    bug_id: str
    success: bool
    winning_attempt: Optional[int] = None
    attempts: List[PatchAttempt] = field(default_factory=list)
    fault_report: Any = None  # Riferimento alla struttura
       FaultReport
    repair_spec: Any = None   # Riferimento alla struttura
       RepairSpec

    def summary(self) -> str:
        """Produce una stringa riassuntiva per l'output a
           terminale."""
        status = "RESOLVED" if self.success else "FAILED"
        winning = f" (Attempt {self.winning_attempt})" if self.
           success else ""
        return f"Bug {self.bug_id}: {status}{winning} - Total
           attempts: {len(self.attempts)}"
```

La struttura `PatchAttempt` abilita una completa tracciabilità post-hoc. Il suo attributo principale è `agent_messages`, che incapsula l'intera storia conversazionale del tentativo: prompt di sistema, invocazioni agli strumenti (con `tool_call_id`), diff diagnostici, output del compilatore e fallimenti JUnit. Questa granularità permette di ricostruire passo dopo passo la catena decisionale del modello, strumento prezioso per il debugging.

A un livello superiore, `RepairResult` è l'artefatto conclusivo aggregato della pipeline a tre fasi: raccoglie la lista dei `PatchAttempt` e ingloba il `FaultReport` (Fase 1) e la `RepairSpec` (Fase 2), unificando diagnosi, specifica e risoluzione. Il metodo `summary()` emette una riga di testo essenziale per il monitoraggio a terminale durante l'esecuzione batch (`run_all_bugs.py`).

Al termine, la persistenza è delegata a `services/reporter.py`, che serializza `RepairResult` in JSON nel file `final_report.json` sotto `pipeline_results/<bug_id>/`. Questo archivio permanente fornisce il dataset grezzo per le successive analisi statistiche di efficacia e i confronti con altri sistemi APR.

3.7.5 Il Flusso Completo: dalla `RepairSpec` al `RepairResult`

Per sintetizzare l'orchestrazione tecnica delineata, si traccia l'intero viaggio dei dati nel `PatchAgent`, che unisce inferenza neurale e oracoli deterministici in tre passaggi. La Figura 3.7 ne illustra l'interazione tra i moduli coinvolti.

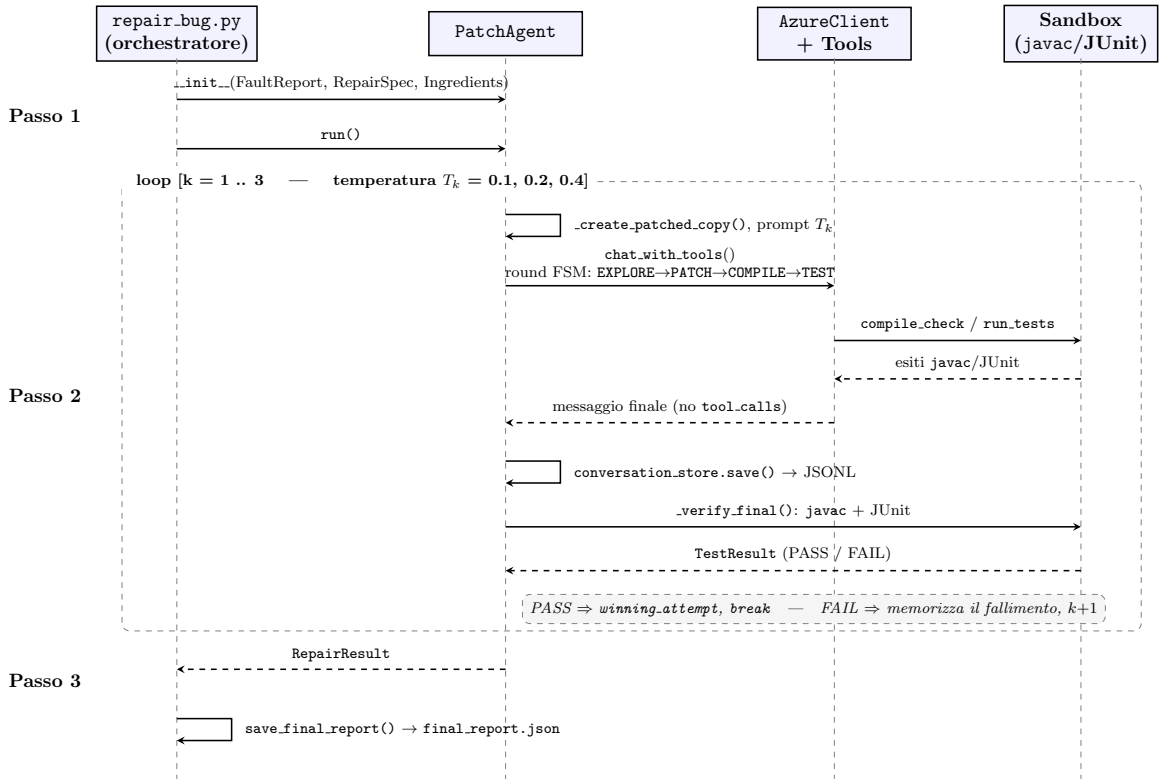

Figura 3.7: Diagramma di sequenza del flusso completo della Fase 3: il viaggio dei dati dai tre artefatti d'ingresso fino al `RepairResult` persistito su disco.

Passo 1, Inizializzazione del Contesto Strategico Nel costruttore `PatchAgent.__init__()` l'agente riceve il trittico di artefatti (`FaultReport`, `RepairSpec`, `Ingredients`), istanzia il client `AzureClient` e pre-alloca un `RepairResult` con `success=False`.

Passo 2, Il Loop della Macchina a Stati Finiti (Max 3 Tentativi) Il cuore operativo concede all'agente un massimo di tre tentativi (`max_attempts = 3`). Per ogni tentativo k:

Isolamento Operativo: `_create_patched_copy()` clona il progetto in una sandbox vergine, azzerando le contaminazioni inter-tentativo.

Modulazione Stocastica: si preleva la temperatura T_k da `LLM_TEMPERATURE_SCHEDULE` (0.1, 0.2, 0.4), aumentando la "creatività" del modello a fronte dei fallimenti.

Innesco del Tool-Use Loop: `_run_single_attempt()` istanzia il `ToolContext`, predispone il dispatcher (`create_tool_executor()`) e crea il prompt con `_build_messages()`; per $k > 1$ il prompt è arricchito coi riassunti dei fallimenti precedenti.

Dinamica Conversazionale: `AzureClient.chat_with_tools()` avvia un loop autonomo (max 9 round, `max_tool_rounds`) in cui il modello si muove tra gli stati FSM (`EXPLORE`, `PATCH`, `COMPILE`, `TEST`), invocando strumenti e manipolando il CST; l'anello si spezza con un responso finale privo di `tool_calls` o all'esaurimento del budget.

Persistenza ed Esame: `conversation_store.save()` serializza lo scambio in JSONL, poi `_verify_final()` invoca `javac` e JUnit sulla sandbox per produrre un `TestResult`.

Valutazione del Verdetto: se l'oracolo restituisce `PASS`, `RepairResult.success` diventa `True`, il tentativo è marcato `winning_attempt` e il ciclo si interrompe; altrimenti l'agente procede al tentativo successivo con la nuova lezione memorizzata.

Passo 3, Restituzione del `RepairResult` Esauriti i tentativi o conseguito il successo, l'oggetto `RepairResult`, ricco di diagnosi, diff, messaggi e metriche, è restituito all'orchestratore `repair_bug.py`, che tramite `reporter.py` lo persiste su disco, sigillando la riparazione e liberando le risorse per il bug successivo.

3.8 Sintesi dell'architettura e limitazioni

Conclusa la descrizione dettagliata delle tre fasi, è utile ricomporre il quadro d'insieme e, soprattutto, discutere con onestà i confini di validità del sistema. Questa sezione finale chiude il capitolo architetturale in due momenti: una sintesi complessiva della *pipeline* e un'analisi critica delle sue principali limitazioni.

3.8.1 Sintesi complessiva della pipeline

L'architettura di `HybridRepair` realizza la separazione tra diagnosi e sintesi illustrata nel Capitolo precedente attraverso tre fasi nettamente disaccoppiate: `FaultOracle` produce una diagnosi causale formale e deterministica; `SpecReason` la trasforma in una specifica di riparazione ragionata, ancorata alle sole API reali; `PatchAgent` traduce infine la specifica in codice, applicandolo e validandolo in modo autonomo. Ciascuna fase è progettata per neutralizzare una specifica classe di fallimento tipica dei sistemi APR basati su LLM.

La Tabella 3.4 riepiloga i tre livelli della Fase 2, associando a ciascun modulo il meccanismo tecnico adottato e la problematica che esso affronta.

Livello	**Componente**	**Meccanismo**	**Problema Affrontato**
Grounding statico	Ingredient Forge	Estrazione regex da sorgente Java	Allucinazione di API
Ragionamento strutturato	SpecBuilder (CoT)	Prompt XML tripartito a bassa temperatura	Deriva semantica e pattern matching
Validazione autonoma	SpecCritic	Auto-critica con confidence scoring	Incongruenza tra specifica ed evidenza empirica

Tabella 3.4: Sintesi del flusso della Fase 2 di HybridRepair.

In modo analogo, la Tabella 3.5 riassume i quattro contributi tecnici che compongono la Fase 3, dalla Macchina a Stati Finiti che struttura il processo fino al sandbox deterministico che ne certifica l'esito.

Contributo	**Modulo**	**Meccanismo**	**Problema Risolto**
FSM di riparazione	`agent.py`	Prompt workflow + retry loop	Mancanza di struttura nel processo generativo
Memoria conversazionale	`conversation_store.py`	Serializzazione JSONL + negative prompting	Loop sterile inter-tentativo
Riparazione strutturale	`ast_applier.py`	Tree-sitter CST + sostituzione byte-offset	Patch rotte da regex sbilanciate
Sandbox deterministico	`sandbox_evaluator.py`	`javac` + `JUnitCore` come oracoli	Validazione sintattica e semantica autonoma

Tabella 3.5: Sintesi dei contributi tecnici della Fase 3 di HybridRepair.

Il prodotto finale della pipeline non è dunque una "supposizione" stocastica del modello, ma una *patch* costruita a partire da evidenza empirica (catene Prolog, test falliti, contesto sintattico), filtrata attraverso un ragionamento strutturato e infine validata da oracoli deterministici. È proprio questo ancoraggio progressivo alla verità del programma a distinguere `HybridRepair` dagli approcci puramente generativi.

3.8.2 Limitazioni

L'architettura, pur introducendo avanzamenti significativi nella gestione del contesto e nel ragionamento guidato per l'*Automated Program Repair*, non è esente da vulnerabilità strutturali e concettuali. Un'analisi rigorosa impone di esaminarne onestamente i limiti operativi, delineando i confini di validità del sistema. Due in particolare meritano attenzione, entrambi originati dalla natura ibrida del sistema.

Dipendenza dalla qualità dell'output di LogicFL

Il primo limite risiede nell'accoppiamento logico tra le fasi della pipeline. L'architettura `SpecReason` incarna a tutti gli effetti un approccio *knowledge-driven* (guidato dalla conoscenza pregressa). Di conseguenza, l'efficacia del paradigma Chain-of-Thought implementato nello `SpecBuilder` è strettamente e inevitabilmente subordinata alla qualità diagnostica dell'output generato dalla Fase 1 (`FaultOracle`). Se il modulo `LogicFL` non riesce a convergere, oppure produce catene causali confuse (scenario frequente in presenza di bug complessi, caratterizzati da propagazioni del valore *null* attraverso molteplici chiamate di metodo *cross-class* o profonde gerarchie di ereditarietà), la sezione {`causal_analysis`} del prompt si rivela scarsamente informativa. In queste circostanze, il sistema subisce un degrado prestazionale severo: il modello linguistico è costretto a dedurre il comportamento difettoso affidandosi quasi unicamente alla lettura del codice sorgente grezzo e dello stack trace. Questo fa collassare il raffinato processo di ragionamento strutturato a un più banale pattern di inferenza generica, rendendo il comportamento di `HybridRepair` indistinguibile da quello dei tradizionali sistemi APR sprovvisti di *fault localization* semantica avanzata. Le evidenze empiriche in letteratura confermano infatti che l'efficacia deduttiva del `CoT` è strettamente vincolata al rapporto segnale-rumore del prompt [40]: i massimi vantaggi del ragionamento strutturato si materializzano appieno solo in presenza di diagnosi `LogicFL` di elevata qualità.

Il limite del self-critique

Il secondo limite riguarda il meccanismo di auto-critica. Il modulo `spec_critic.py`, pur mitigando le allucinazioni più palesi, soffre di un bias cognitivo sistematico ben documentato nella letteratura inerente alla *self-evaluation* dei Large Language Models [55]. Il nocciolo del problema risiede nel fatto che un modello linguistico incontra enormi difficoltà nell'identificare e correggere errori logici o deduttivi che esso stesso ha la tendenza a commettere sistematicamente [15].

Essendo valutatore e generatore istanze del medesimo modello base (`GPT-4o`), essi condividono le stesse "zone d'ombra" euristiche e parametriche. Se, per esempio, lo `SpecBuilder` deduce una specifica errata scambiando regolarmente una causa distale per una causa prossima all'interno di una catena NPE complessa, lo `SpecCritic` tenderà con alta probabilità a validare tale ragionamento fallace, assegnandogli un elevato punteggio di confidenza. Questo fenomeno, definito *echo chamber effect* dell'autovalutazione [30], si innesca perché il modello validatore riconosce e premia il pattern deduttivo che esso stesso avrebbe generato. Per spezzare questo bias di conferma speculare, la validazione necessiterebbe di un orizzonte di verifica esterno e strutturalmente indipendente: le possibili contromisure, validazione *cross-model*, *Process Reward Model* e feedback esecutivo da oracoli deterministici, sono discusse tra le direzioni di miglioramento (Capitolo 5).

Capitolo 4

Sperimentazione

4.1 Obiettivi

L'obiettivo principale è quello misurare in modo rigoroso la capacità del sistema di generare automaticamente patch plausibili e patch valide per bug di tipo NullPointerException reali, estratti dal benchmark Defects4J. Il secondo obiettivo è confrontare le prestazioni di HybridRepair con quelle di VibeRepair, un sistema APR allo stato dell'arte basato esclusivamente su Large Language Models, che a differenza di HybridRepair si affida alla Perfect Fault Localization, ovvero assume come noto a priori il punto esatto del bug, anziché eseguire una localizzazione causale autonoma tramite analisi logica.

4.2 Research Questions

La sperimentazione è strutturata attorno a tre domande di ricerca:

- **RQ1:** Quante patch plausibili è in grado di generare HybridRepair sui bug NPE del dataset Defects4J?
- **RQ2:** Quante delle patch plausibili generate da HybridRepair sono anche valide, ovvero semanticamente equivalenti o sostanzialmente equivalenti al Ground Truth?
- **RQ3:** Come si confrontano le prestazioni di HybridRepair rispetto a VibeRepair in termini di patch valide generate sullo stesso dataset di bug NPE?

4.3 Metriche

Prima di introdurre le metriche è necessario definire con precisione i due concetti su cui si basano, ripresi dalla letteratura sull'Automated Program Repair [32]:

- **Patch plausibile:** una patch generata automaticamente che compila correttamente e supera l'intera suite di test del bug (test di regressione inclusi), senza alcuna garanzia di correttezza semantica al di fuori dei casi coperti dai test.

- **Patch valida:** una patch plausibile che, a seguito di revisione manuale, risulta semanticamente equivalente o sostanzialmente equivalente alla patch di riferimento (Ground Truth) del bug, ovvero risolve effettivamente la causa del difetto anziché mascherarne i sintomi.

Sulla base di queste due nozioni sono state definite le seguenti metriche:

Sia N il numero totale di bug nel dataset, P_{HR} il numero di bug per cui HybridRepair produce almeno una patch plausibile, V_{HR} il numero di patch plausibili di HybridRepair giudicate valide, e C_{VR} il numero di patch valide prodotte da VibeRepair.

- **Patch Plausibili (HR Plaus.):** rapporto tra il numero di bug per cui il sistema produce almeno una patch che supera la suite di test e il numero totale di bug nel dataset. Corrisponde alla metrica standard adottata in letteratura per la valutazione dei sistemi APR.

$$\text{HR Plaus.} = \frac{P_{HR}}{N}$$

- **Patch Valide (HR Valida):** rapporto tra il numero di patch plausibili giudicate valide tramite revisione manuale e il numero totale di patch plausibili. Questa metrica quantifica la precision del sistema, ovvero la capacità di generare soluzioni semanticamente corrette tra quelle che superano la build.

$$\text{HR Valida} = \frac{V_{HR}}{P_{HR}}$$

- **Tasso di patch valide relativo (Plausibile AND Valida HR):** rapporto tra il numero di patch valide e il numero totale di bug nel dataset. A differenza di HR Valida, il denominatore è l'intero dataset e non le sole patch plausibili; questa metrica viene utilizzata come base omogenea di confronto con VibeRepair, per cui non sono disponibili dati sul numero di patch plausibili generate.

$$\text{Plausibile AND Valida HR} = \frac{V_{HR}}{N}$$

- **Tasso di patch valide relativo VibeRepair (Plausibile AND Valida VR):** rapporto tra il numero di patch corrette di VibeRepair e il numero totale di bug nel dataset. È la metrica analoga a Plausibile AND Valida HR applicata a VibeRepair; i valori sono estratti dai dati pubblicati nel paper originale [57] e rappresentano la base omogenea di confronto con HybridRepair.

$$\text{Plausibile AND Valida VR} = \frac{C_{VR}}{N}$$

La distinzione tra le prime due metriche è fondamentale nel contesto dell'Automated Program Repair. Come evidenziato in letteratura, il fenomeno della test-suite blindness, ovvero la generazione di patch che passano i test esistenti pur introducendo regressioni o soluzioni semanticamente errate, è una delle criticità principali dei sistemi APR. Una patch plausibile può mascherare il bug senza risolverlo, violando contratti API non esplicitati nei test o introducendo comportamenti non corretti su casi limite non coperti dalla suite. La metrica HR Valida quantifica esattamente la capacità del sistema di andare oltre la plausibilità superficiale.

4.4 Oggetti Sperimentali

Il dataset sperimentale è composto da 37 bug di tipo NullPointerException estratti dal benchmark Defects4J v2. La selezione è stata effettuata scegliendo bug NPE provenienti da progetti Java open source di media e grande dimensione.

Il dataset copre 12 progetti distinti del benchmark Defects4J:

- **JFreeChart (Chart):** libreria per la generazione di grafici e diagrammi in Java.
  `Chart-2, Chart-4, Chart-14, Chart-16`
- **Apache Commons CLI (Cli):** libreria per la gestione di interfacce a riga di comando in Java.
  `Cli-5, Cli-30`
- **Google Closure Compiler (Closure):** strumento per l'ottimizzazione e la compilazione di codice JavaScript.
  `Closure-2, Closure-171`
- **Apache Commons Codec (Codec):** libreria per la codifica e decodifica di dati in vari formati, come Base64 e URL encoding.
  `Codec-5, Codec-13, Codec-17`
- **Apache Commons CSV (Csv):** libreria per la lettura e scrittura di file CSV in Java.
  `Csv-4, Csv-9, Csv-11`
- **Google Gson (Gson):** libreria per la serializzazione e deserializzazione di oggetti Java in formato JSON.
  `Gson-6, Gson-9`
- **FasterXML Jackson Core (JacksonCore):** libreria per la manipolazione di dati JSON.
  `JacksonCore-8`
- **FasterXML Jackson Databind (JacksonDatabind):** libreria per la serializzazione e deserializzazione di oggetti Java in formato JSON.
  `JacksonDatabind-3, JacksonDatabind-13, JacksonDatabind-80, JacksonDatabind-93, JacksonDatabind-95, JacksonDatabind-107`
- **jsoup (Jsoup):** libreria per l'analisi e la manipolazione di documenti HTML in Java.
  `Jsoup-8, Jsoup-22, Jsoup-26, Jsoup-66, Jsoup-89`
- **Apache Commons Lang (Lang):** libreria che fornisce utilità per la manipolazione di stringhe, numeri e oggetti in Java.
  `Lang-20, Lang-33, Lang-39, Lang-47, Lang-57`
- **Apache Commons Math (Math):** libreria per la matematica e la statistica in Java.
  `Math-4, Math-70, Math-79`
- **Mockito:** framework per il testing unitario in Java, utilizzato per creare oggetti mock e simulare comportamenti di dipendenze esterne durante i test.
  `Mockito-18`

4.5 Materiale Sperimentale

4.5.1 HybridRepair

Il sistema HybridRepair presenta una pipeline composta da tre componenti principali, descritti in dettaglio nel Capitolo dell'Architettura.

Il primo componente è **LogicFL** [22], uno strumento di fault localization basato su inferenza logica in Prolog, che identifica la causa radice di un NPE tracciando la propagazione del valore `null` attraverso il codice sorgente. LogicFL è stato eseguito nella versione disponibile nel replication package ufficiale degli autori, con SWI-Prolog come motore di inferenza.

Il secondo componente è il modulo di **prompt generativi** (SpecReason e PatchAgent), che utilizza il modello GPT-4o (deployment Azure `gpt-4o`, API version `2024-02-01`) come modello linguistico di base, accessibile tramite Azure OpenAI API. La manipolazione strutturale del codice sorgente è implementata tramite Tree-sitter. Ogni bug è stato eseguito con un budget massimo di 3 tentativi, con un massimo di 9 round di tool-use per tentativo. I tre tentativi non sono esecuzioni indipendenti, ma vengono eseguiti in sequenza all'interno della stessa sessione di interazione con il modello: come descritto nel Capitolo dell'Architettura (meccanismo di *retry* adattivo), il riassunto del tentativo fallito viene iniettato come contesto negativo in quello successivo, per cui ogni nuovo tentativo dispone di un contesto via via più ampio rispetto al precedente.

Il terzo componente è il **valutatore automatico di plausibilità**, che verifica il superamento di tutti i test da parte della patch generata eseguendo la suite di test JUnit ufficiale di ciascun progetto.

4.5.2 VibeRepair

VibeRepair [57] è un sistema APR basato esclusivamente su Large Language Models che adotta un approccio *specification-centric*: anziché modificare direttamente il codice sorgente, traduce il codice difettoso in una specifica comportamentale, corregge la specifica, e genera la patch a partire dalla specifica che lui ritiene corretta. A differenza di HybridRepair, VibeRepair non esegue una localizzazione del fault autonoma, bensì opera assumendo la *fault localization* tramite *Perfect Fault Localization* (PFL), ovvero assume come noto a priori il punto esatto del bug. Per la valutazione su Defects4J, VibeRepair utilizza GPT-4o (`gpt-4o-2024-05-13`) come modello di base, con temperatura pari a 1 e un budget massimo di $5 \times 3 = 15$ patch candidate per bug.

I dati di VibeRepair utilizzati per il confronto sono quelli pubblicati nel paper originale [57] per Defects4J v2.0. Il confronto è quindi di tipo archivio contro archivio: per ciascuno dei 37 bug del dataset, è stato verificato se VibeRepair disponeva di una patch plausibile per lo stesso identificatore di bug. Si noti che VibeRepair è stato valutato su un insieme più ampio di bug (438 bug totali in Defects4J v2.0): il confronto è quindi ristretto al sottoinsieme dei 37 bug di categoria NPE del presente dataset.

4.5.3 Dataset

I bug NPE utilizzati nella sperimentazione sono stati estratti da Defects4J v2.0.1 [20]. Il dataset è stato ottenuto tramite il replication package di LogicFL, che include uno

snapshot preconfigurato dei bug NPE selezionati dagli autori, comprensivo delle versioni difettose dei progetti e delle suite di test JUnit necessarie per la riproduzione degli errori. Ogni bug è stato eseguito nella versione difettosa del progetto, con la suite di test JUnit ufficiale come oracolo di validazione per la plausibilità delle patch.

4.5.4 Ambiente di Esecuzione

L'intera pipeline è stata eseguita su una macchina con sistema operativo Ubuntu 24.04.3 LTS. Le versioni dei principali componenti software utilizzati sono le seguenti:

- **Java:** OpenJDK 11.0.31, utilizzato sia per l'esecuzione di LogicFL sia per la compilazione e l'esecuzione delle suite di test JUnit dei progetti Defects4J tramite Apache Maven;
- **Apache Maven:** 3.8.7, utilizzato per la gestione delle dipendenze e l'esecuzione delle build dei progetti;
- **Python:** 3.12.3, utilizzato per l'orchestrazione della pipeline HybridRepair (FaultOracle, SpecReason, PatchAgent) e per la manipolazione del codice sorgente tramite Tree-sitter (`tree-sitter` 0.25.2, `tree-sitter-java` 0.23.5); le chiamate ad Azure OpenAI sono effettuate tramite la libreria `openai` 2.40.0;
- **SWI-Prolog:** 9.0.4 con JPL, motore di inferenza utilizzato da LogicFL per la fault localization basata su regole logiche.

4.6 Procedura Sperimentale

4.6.1 Esecuzione della pipeline HybridRepair

Per ciascun bug del dataset, la pipeline è stata eseguita in modalità fully-automated: dato l'identificatore del bug (ad esempio Chart-4), il sistema ha orchestrato in sequenza le tre fasi (FaultOracle, SpecReason, PatchAgent) producendo in output un oggetto RepairResult. I risultati sono stati classificati come: PASS se almeno uno dei tre tentativi ha prodotto una patch che supera la suite di test completa; FAIL se nessun tentativo ha superato la build.

4.6.2 Protocollo di validazione manuale

La determinazione della validità di una patch ha richiesto una revisione manuale per ciascuna delle 22 patch plausibili, condotta con il supporto di strumenti di analisi automatica del codice per l'ispezione statica dei diff. La procedura adottata è stata la seguente:

- **Confronto della patch generata con il Ground Truth** ufficiale di Defects4J riga per riga.
- **Verifica dell'equivalenza semantica**, ovvero se la patch risolve la stessa causa radice del Ground Truth e produce lo stesso comportamento osservabile su tutti gli input, compresi i casi limite.

- **Verifica dell'assenza di regressioni silenti**, tramite ispezione statica del diff e ragionamento sui code path interessati dalla modifica, per identificare eventuali divergenze di comportamento su scenari non coperti dalla suite di test.
- Nei casi *borderline*, in cui la divergenza dal Ground Truth D4J non fosse attribuibile con certezza a una lacuna di HybridRepair, confronto con le patch di riferimento pubblicate da VibeRepair, come calibrazione indipendente del giudizio.

Una patch è stata classificata come valida se soddisfa almeno uno dei seguenti criteri:

- **Equivalenza** letterale con il **Ground Truth**, con differenze limitate a variazioni stilistiche non semantiche.
- **Differenza funzionale** con strategia di fix diversa, purché la patch produca lo stesso output osservabile della GT sugli input della suite di test e sui casi limite individuati durante l'ispezione manuale del diff, senza regressioni rilevabili nei code path modificati (cfr. protocollo di validazione, Sezione 4.6.2).
- **Convergenza** con la patch di VibeRepair su tutti i metodi rilevanti: in accordo con la prassi consolidata nella ricerca APR, una patch giudicata valida in un sistema peer-reviewed pubblicato costituisce un oracolo indipendente. Se HybridRepair produce la medesima soluzione, questa è considerata valida per transitività, nei casi in cui la divergenza dal Ground Truth D4J non sia penalizzata da alcun test della suite.

Una patch è stata classificata come plausibile e non valida se non soddisfa nessuno dei criteri di validità sopra elencati.

4.6.3 Estrazione delle patch di VibeRepair

Per il confronto con HybridRepair, le patch valide di VibeRepair sono state estratte dai dati pubblici del paper originale, specificamente dai file delle patch semanticamente equivalenti al Ground Truth. Per ciascuno dei 37 bug del dataset, è stato verificato se VibeRepair disponeva di una patch corretta per lo stesso identificatore di bug. Il confronto è quindi di tipo archivio-contro-archivio sulla metrica delle patch valide e valide, non una riesecuzione diretta sullo stesso ambiente.

4.7 Risultati

La Tabella riporta i risultati per ciascun bug del dataset secondo le metriche definite nella Sezione 4.3: **HR Plaus.**, **HR Valida** e **Plausibile AND Valida HR** per HybridRepair; **Plausibile AND Valida VR** per VibeRepair[57]. Il simbolo ✓ indica esito positivo, × esito negativo, − indica che la patch non è plausibile (la validità non è applicabile).

	HybridRepair			**VibeRepair**
Bug ID	**HR Plaus.**	**HR Valida**	**Plausibile AND Valida HR**	**Plausibile AND Valida VR**
Chart-2	×	−	×	×
Chart-4	✓	✓	✓	×
Chart-14	✓	✓	✓	×
Chart-16	✓	×	×	×
Cli-5	✓	✓	✓	✓
Cli-30	×	−	×	✓
Closure-2	✓	×	×	×
Closure-171	×	−	×	×
Codec-5	×	−	×	✓
Codec-13	✓	✓	✓	×
Codec-17	✓	✓	✓	×
Csv-4	✓	✓	✓	✓
Csv-9	✓	×	×	✓
Csv-11	✓	×	×	✓
Gson-6	✓	✓	✓	✓
Gson-9	×	−	×	×
JacksonCore-8	✓	×	×	✓
JacksonDatabind-3	✓	✓	✓	✓
JacksonDatabind-13	✓	×	×	×
JacksonDatabind-80	✓	✓	✓	✓
JacksonDatabind-93	✓	✓	✓	✓
JacksonDatabind-95	✓	✓	✓	✓
JacksonDatabind-107	×	−	×	×
Jsoup-8	✓	✓	✓	×
Jsoup-22	×	−	×	×
Jsoup-26	✓	✓	✓	✓
Jsoup-66	×	−	×	×
Jsoup-89	✓	×	×	✓
Lang-20	✓	✓	✓	×
Lang-33	×	−	×	×
Lang-39	✓	✓	✓	×
Lang-47	×	−	×	×
Lang-57	×	−	×	×
Math-4	×	−	×	×
Math-70	×	−	×	×
Math-79	×	−	×	×
Mockito-18	×	−	×	×
Totale	**22/37 (59,5%)**	**15/22 (68,2%)**	**15/37 (40,5%)**	**14/37 (37,8%)**

4.8 RQ1, Patch plausibili

HybridRepair genera patch plausibili per **22 bug su 37 (59,5%)**. Il sistema ha fallito su 15 bug. I fallimenti si concentrano prevalentemente su progetti con strutture di propagazione del valore `null` molto complesse o su bug con dipendenze inter-procedurali profonde non completamente risolte da LogicFL (Closure-171, JacksonDatabind-107, i bug di Commons Math e Mockito).

> **Risposta a RQ1:** HybridRepair genera patch plausibili per il 59,5% dei bug NPE testati (22/37). Il numero di patch plausibili di VibeRepair sui 37 bug del dataset non è direttamente ricavabile dai dati pubblicati.

4.9 RQ2, Patch valide

Delle 22 patch plausibili, **15 sono state giudicate valide (68,2% di precision)**, ovvero semanticamente equivalenti o sostanzialmente equivalenti al Ground Truth.

> **Risposta a RQ2:** Il 68,2% delle patch plausibili di HybridRepair è semanticamente valido (15/22), corrispondente al 40,5% del dataset complessivo (15/37).

4.10 RQ3, Confronto con VibeRepair

Il confronto con VibeRepair è effettuato sulla metrica delle **patch valide**, che rappresenta l'unica base di confronto omogenea disponibile: i dati pubblicati da VibeRepair [57] riportano patch valide, non patch meramente plausibili. La tabella indica la distribuzione dei bug in base all'esito combinato dei due sistemi su questa metrica.

Scenario	**N. bug**	**Bug**
Solo HybridRepair produce patch valida	7	Chart-4, Chart-14, Codec-13, Codec-17, Jsoup-8, Lang-20, Lang-39
Solo VibeRepair produce patch corretta	6	Cli-30, Codec-5, Csv-9, Csv-11, JacksonCore-8, Jsoup-89
Entrambi producono patch valida	8	Cli-5, Csv-4, Gson-6, JacksonDatabind-3, JacksonDatabind-80, JacksonDatabind-93, JacksonDatabind-95, Jsoup-26
Nessuno dei due	16	Chart-2, Chart-16, Closure-2, Closure-171, Gson-9, JacksonDatabind-13, JacksonDatabind-107, Jsoup-22, Jsoup-66, Lang-33, Lang-47, Lang-57, Math-4, Math-70, Math-79, Mockito-18
Totale	**37**	

HybridRepair produce **15/37 patch valide (40,5%)**, contro le **14/37 (37,8%)** di VibeRepair sullo stesso dataset. Tuttavia, il confronto nasconde una differenza fondamentale nelle condizioni operative: VibeRepair, come abbiamo accennato precedentemente, opera sotto l'assunzione di *Perfect Fault Localization*, mentre HybridRepair esegue la localizzazione del fault in modo autonomo tramite LogicFL, senza alcuna informazione a priori sulla posizione del difetto. Superare VibeRepair in termini di patch valide, operando in condizioni realistiche senza PFL, rappresenta il risultato principale di questo confronto.

> **Risposta a RQ3:** HybridRepair produce patch valide per 15/37 bug (40,5%), superando le 14/37 di VibeRepair sullo stesso dataset (37,8%). Il risultato è notevole in quanto HybridRepair opera senza Perfect Fault Localization a differenza di VibeRepair. I due sistemi sono complementari: HybridRepair risolve 7 bug che VibeRepair non corregge, VibeRepair ne risolve 6 che HybridRepair non corregge, con una sovrapposizione di 8 bug risolti da entrambi.

4.11 Discussione

Le 15 patch valide presentano pattern caratteristici: fix minimali e chirurgici identici al Ground Truth a meno di variazioni stilistiche (Chart-4, Chart-14, Lang-39, JacksonDatabind-80, JacksonDatabind-95, Csv-4), ed equivalenze funzionali con stile di codifica diverso ma comportamento identico (Jsoup-8, Gson-6, Lang-20, Codec-13, Codec-17, JacksonDatabind-3). Questo risultato dimostra che la guida semantica di LogicFL, combinata con il paradigma Chain-of-Thought di SpecReason, orienta il modello generativo verso la causa radice del bug anziché verso patch superficiali.

Analizzando i bug risolti esclusivamente da ciascun sistema, qui invece emergono due pattern distinti. HybridRepair eccelle su bug multi-punto, ovvero bug che richiedono modifiche in più punti distinti del codice sorgente, con propagazione NPE cross-method (Chart-4, Chart-14, Codec-13, Codec-17, Lang-39), dove la catena causale di LogicFL guida il modello verso tutti i punti di intervento necessari. VibeRepair produce invece patch valide su bug dove la specifica comportamentale è più facilmente inferibile dai test fallenti (Csv-9, Csv-11, JacksonCore-8, Jsoup-89). Su questi stessi bug, HybridRepair genera patch plausibili che superano i test ma presentano una correzione parziale del difetto o introducono comportamento scorretto su aspetti non testati dalla suite, un caso tipico di *test-suite blindness*, e sono state quindi classificate come non valide.

La dipendenza dell'intera pipeline dall'output di LogicFL rappresenta inoltre una causa diretta di peggioramento delle prestazioni: su alcuni bug LogicFL non converge o produce catene causali incomplete, degradando le prestazioni di SpecReason e PatchAgent a quelle di un sistema APR senza fault localization semantica. La pipeline non si blocca in presenza di output incompleto — SpecReason opera comunque sul contesto disponibile — ma ciò si traduce in un degrado graduale delle prestazioni piuttosto che in un fallimento netto dell'intero sistema.

4.11.1 Casi di successo rappresentativi

Chart-14 rappresenta un caso emblematico dell'efficacia dell'approccio ibrido su bug multi-punto. Il fix richiede l'inserimento della stessa guard clause in quattro metodi gemelli distribuiti su due classi distinte, per dominio e range. HybridRepair identifica correttamente tutti e quattro i punti di intervento grazie alla catena causale prodotta da LogicFL, che traccia la propagazione del valore null attraverso i quattro percorsi di esecuzione. VibeRepair, pur operando con Perfect Fault Localization, non produce patch corretta per questo bug.

Lang-39 illustra la capacità del sistema di gestire bug con origini di nullità in parametri di array. La patch generata introduce una guard clause per saltare le coppie di elementi null nei due array di input, semanticamente identica al Ground Truth; l'unica differenza è l'ordine dei due operandi della condizione, una variazione stilistica priva di effetti semantici.

Csv-4 evidenzia il funzionamento del meccanismo di retry adattivo. Il bug si manifesta quando il parser è configurato senza header: il tentativo di costruire una copia difensiva di una mappa nulla solleva un NPE, mentre il contratto prevede la restituzione di `null`. Il primo tentativo produceva una soluzione eccessivamente ampia, modificando più punti del codice e causando la restituzione di una mappa vuota invece di `null`, con il fallimento di 1 test su 48. Il secondo tentativo ha generato una guard clause ternaria di una sola riga quasi identica al Ground Truth, con l'unica differenza in una variazione sintattica stilistica senza effetti semantici.

JacksonDatabind-3 è l'esempio migliore del valore nel confronto con sistemi indipendenti (VibeRepair) nella classificazione dei casi limite. Il bug riguarda la gestione di un token null all'interno di un deserializzatore: HybridRepair introduce una guardia esplicita sull'oggetto deserializzatore che, grazie a un controllo analogo già presente a monte nel flusso, produce sempre lo stesso risultato del Ground Truth. La patch era stata inizialmente classificata come caso di test-suite blindness per una divergenza su un metodo secondario dello stesso file, dove il Ground Truth introduce una modifica aggiuntiva che HybridRepair omette. Il confronto con VibeRepair ha però rivelato che anche quest'ultimo adotta la stessa scelta di HybridRepair su quel metodo secondario, anziché seguire il Ground Truth. La divergenza è dunque una scelta indipendentemente condivisa da VibeRepair, sistema peer-reviewed, non penalizzata da alcun test nella suite: in base al criterio di convergenza definito nel protocollo di validazione, la patch è stata classificata come valida.

4.11.2 Casi problematici

Le 7 patch plausibili-non-valide di HybridRepair si distribuiscono in due sottocategorie: 2 casi intercettati automaticamente dal validatore semantico della pipeline (Csv-9, Csv-11), e 5 casi di *test-suite blindness* (l'incapacità della suite di test di rilevare che una patch, pur superandola, non risolve correttamente il difetto o introduce regressioni non coperte) rilevati solo dalla revisione manuale (Chart-16, Closure-2, JacksonCore-8, JacksonDatabind-13, Jsoup-89).

Csv-9 e Csv-11 rappresentano i due casi intercettati dal validatore semantico automatico. In entrambi, la patch supera la suite di test ma viola un contratto API non coperto dai test stessi. In Csv-9, il modello è intervenuto nel metodo sbagliato: ha cor-

retto il punto dove il valore null si manifestava, anziché la sua origine, alterando così il contratto documentato relativo al comportamento atteso in assenza di intestazione. In Csv-11, la patch ha introdotto un'istruzione di skip sulle colonne con intestazione nulla che, pur eliminando l'eccezione, ha disallineato la corrispondenza tra indici numerici e nomi di colonna, alterando silenziosamente il comportamento dell'intera struttura dati. Questi casi confermano l'utilità del validatore semantico come filtro preventivo rispetto alla sola esecuzione della suite di test.

Chart-16 illustra un caso di over-editing. La patch corregge correttamente il costruttore, ma interviene anche su un metodo correlato rimuovendo sia un controllo di validazione sia una notifica di aggiornamento agli osservatori registrati. La rimozione introduce una regressione comportamentale non rilevata dai test, poiché la notifica rimossa ha effetti collaterali attesi su altri componenti del sistema. Questo evidenzia come il vincolo di minimalità nel prompt debba essere bilanciato con una comprensione degli invarianti del codice: non è sufficiente minimizzare le righe modificate se non si preservano le operazioni con effetti collaterali.

4.12 Threats to Validity

4.12.1 Validità interna

- **Soggettività nella validazione manuale:** la classificazione di una patch come valida o come plausibile e non valida richiede una valutazione dell'equivalenza semantica con il Ground Truth che, sebbene guidata da criteri espliciti definiti a priori, introduce un margine di soggettività nella valutazione dei casi limite. Per mitigare questo rischio, nei casi di incertezza è stato adottato un criterio conservativo: in assenza di evidenza sufficiente di equivalenza semantica, la patch è stata classificata come non valida, garantendo che eventuali errori di classificazione vadano nella direzione di sottostimare le prestazioni del sistema piuttosto che sovrastimarle.
- **Non-determinismo del LLM:** le risposte del modello linguistico sono stocastiche, pertanto esecuzioni ripetute dello stesso bug potrebbero produrre patch diverse, e i risultati qui riportati provengono da un'unica esecuzione dell'intera pipeline sul dataset. Il budget di 3 tentativi per bug, con schedule di temperatura crescente (0.1, 0.2, 0.4), non mitiga questa minaccia: è una scelta di design orientata a diversificare i tentativi per aumentare il recall sul singolo bug, non un meccanismo statistico di riduzione della varianza sulle metriche aggregate. Una mitigazione rigorosa richiederebbe temperatura nulla, incompatibile però con lo schedule adattivo adottato per uscire dai fallimenti, oppure la replicazione dell'intero esperimento su più run indipendenti (cfr. Capitolo 5); nessuna delle due è stata applicata in questo lavoro.

4.12.2 Validità esterna

- **Dimensione del dataset:** il dataset di 37 bug, pur coprendo 12 progetti Java open source eterogenei, è limitato a bug di tipo NPE estratti da Defects4J. I pattern di successo osservati, in particolare l'efficacia su bug multi-punto con propagazione cross-method, potrebbero non trasferirsi a categorie di difetti struttu-

ralmente diverse o a basi di codice con dipendenze inter-procedurali più profonde di quelle rappresentate nel dataset.

- **Potenziale data leakage del LLM:** i bug di Defects4J potrebbero essere presenti nei dati di pre-training di GPT-4o, introducendo un rischio di memorizzazione del Ground Truth. Tuttavia, poiché sia HybridRepair che VibeRepair utilizzano lo stesso modello base, l'effetto è simmetrico e non influenza le conclusioni comparative di RQ3: eventuali vantaggi da memorizzazione si applicano in egual misura a entrambi i sistemi, rendendo la differenza di prestazioni attribuibile all'architettura e non al modello sottostante.

4.12.3 Validità di costrutto

- **Definizione di "patch valida":** la scelta del Ground Truth di Defects4J come riferimento per la validità introduce un bias: il Ground Truth rappresenta la soluzione scelta dagli sviluppatori originali, ma potrebbero esistere patch semanticamente corrette e alternative equivalenti. Alcune patch classificate come plausibili e non valide potrebbero essere in realtà soluzioni alternative ugualmente corrette. È tuttavia importante notare la direzione di questo bias: eventuali errori di classificazione porterebbero a sottostimare le prestazioni di HybridRepair, non a sovrastimarle, rendendo i risultati riportati conservativi rispetto alle possibili capacità reali del sistema.
- **Confronto asimmetrico con VibeRepair:** i risultati di VibeRepair utilizzati nel confronto provengono dal paper originale e non da una riesecuzione controllata sullo stesso ambiente sperimentale. Differenze nella versione del modello GPT-4o utilizzato (HybridRepair usa gpt-4o-2024-02-01, VibeRepair usa gpt-4o-2024-05-13) potrebbero influenzare il confronto. La versione utilizzata da HybridRepair non è stata una scelta discrezionale, bensì l'unica disponibile tramite il deployment Azure OpenAI al momento della sperimentazione. In particolare, la versione più recente di GPT-4o utilizzata da VibeRepair costituisce un vantaggio strutturale a suo favore: il fatto che HybridRepair superi comunque VibeRepair in queste condizioni rende il risultato comparativo più robusto.

Capitolo 5

Direzioni di miglioramento

L'analisi suggerisce cinque aree di intervento per una successiva iterazione del sistema, dalla sintesi delle patch fino all'estensione della valutazione empirica.

5.1 Controlli mirati sulla sintesi

I casi problematici indicano tre interventi puntuali. I due casi intercettati dal validatore semantico (Csv-9, Csv-11) suggeriscono di estendere il valutatore di plausibilità con un controllo automatico dei contratti documentati nel Javadoc (es. `@return`, `@throws`), affiancato da un'analisi di nullità basata su *pluggable type-checking* [31], per intercettare a monte le violazioni di contratto non coperte dai test. Il caso Closure-2 suggerisce di estendere `SpecReason` con una verifica esplicita degli effetti dell'*early return* sui rami di controllo preesistenti (es. chiamate ricorsive). Il caso Chart-16 indica infine di arricchire il prompt del `PatchAgent` con un'analisi statica delle chiamate rimosse, così da segnalare come potenziale regressione la rimozione di chiamate con effetti collaterali, anche quando la patch supera i test.

5.2 Validazione esterna e indipendente

Il margine più rilevante riguarda l'auto-critica dello `SpecCritic` che, usando lo stesso modello come generatore e valutatore, è soggetto a un bias di conferma speculare (cfr. Sezione 3.8.2). Spezzarlo richiede orizzonti di verifica esterni, lungo tre direttrici. La prima è il **Multi-Agent Debate** (valutazione Cross-Model) [12]: affidare la critica a un modello di un'altra famiglia (es. Claude o Gemini) sfrutta l'eterogeneità di pre-training e architettura affinché i *blind-spot* di generatore e validatore non coincidano. La seconda è un **Process Reward Model (PRM)** [26], specializzato via fine-tuning nella verifica di ogni singolo step della `Chain-of-Thought` anziché nella generazione di codice. La terza ancora la critica a oracoli deterministici (*execution-based feedback*) [36]: *type checker* che intercettano a compile-time le allucinazioni (es. variabili non dichiarate) e *model checker* formali capaci di provare l'assenza di *deadlock*.

5.3 Rafforzamento dell'oracolo di test

La causa prima delle 7 patch plausibili-non-valide è la debolezza della suite come oracolo: una patch che supera i test può violare il comportamento atteso su casi non coperti, fenomeno noto come *overfitting* alla suite di test [37]. Oltre a rafforzare il giudizio semantico, si può irrobustire l'oracolo stesso, generando automaticamente nuovi test che ne amplino la copertura comportamentale prima del giudizio di plausibilità [48]. Due tecniche sono promettenti: il **differential testing**, che confronta la patch candidata con il programma di riferimento, o con patch candidate alternative, su input generati per esporre divergenze osservabili, e il **mutation testing** [18], che misura quanto la suite distingue varianti comportamentali, individuando le regioni sotto-specificate come i metodi secondari coinvolti nei casi Chart-16 e JacksonDatabind-13.

5.4 Robustezza della fault localization

I 15 fallimenti di RQ1 si concentrano su bug con catene causali complesse o dipendenze inter-procedurali profonde (Closure-171, JacksonDatabind-107, i bug di Commons Math e Mockito-18): migliorare LogicFL su questi pattern, o introdurre un fallback che combini più ipotesi di localizzazione, potrebbe aumentare la copertura del sistema. La stessa direzione mitigherebbe anche il degrado graduale delle prestazioni di SpecReason e PatchAgent osservato quando LogicFL non converge o produce catene causali incomplete (cfr. Sezione 4.11), poiché un fallback multi-ipotesi fornirebbe un contesto causale più solido su cui operare anche nei casi in cui l'output primario di LogicFL è parziale.

5.5 Replicazione multi-run

I risultati riportati provengono da un'unica esecuzione della pipeline sul dataset, e il non-determinismo del LLM rimane una minaccia non mitigata (cfr. Sezione 4.12). Una direzione naturale è replicare l'intero esperimento su più run indipendenti, riportando per ciascuna metrica una distribuzione anziché un singolo valore puntuale. Questo richiede però di definire a priori una politica per i bug con esiti discordanti tra le repliche (ad es. plausibile in alcune run e non in altre): tra le opzioni possibili, un criterio di maggioranza sulle repliche, oppure il riporto esplicito dell'intervallo di variazione osservato per ciascun bug invece di un singolo esito binario.

5.6 Estensione del dataset

Una successiva sperimentazione dovrebbe estendere il dataset oltre i 37 bug NPE qui considerati, per verificare se i pattern di successo osservati (bug multi-punto con propagazione cross-method) si confermano su un campione più ampio e su altre categorie di difetti.

Capitolo 6

Conclusioni

Questa tesi è nata da un problema concreto dell'*Automated Program Repair*: i sistemi basati esclusivamente su Large Language Models generano patch a partire da una localizzazione del difetto assunta come nota, mentre gli approcci puramente analitici faticano a sintetizzare il codice della correzione. HybridRepair unisce i due mondi, facendo precedere la generazione della patch da una localizzazione causale autonoma (LogicFL) e da una guida semantica (SpecReason), così da orientare il modello verso la causa radice del bug anziché verso correzioni superficiali.

La sperimentazione su 37 bug NullPointerException di Defects4J conferma la validità dell'approccio. HybridRepair genera patch plausibili per il 59,5% dei bug e patch semanticamente valide per il 40,5%, superando VibeRepair sullo stesso dataset (37,8%). Il contributo principale non sta però nel margine numerico, bensì nelle condizioni in cui è ottenuto: HybridRepair raggiunge questo risultato *senza* Perfect Fault Localization, localizzando il difetto in autonomia, mentre VibeRepair assume noto a priori il punto del bug. L'analisi qualitativa mostra inoltre che i due sistemi sono complementari e che la guida causale rende HybridRepair particolarmente efficace sui bug multi-punto con propagazione *cross-method*, dove la sola inferenza dai test fallenti non basta.

I limiti del lavoro e le possibili estensioni sono discussi rispettivamente nella Sezione 4.12 e nel Capitolo 5: il rafforzamento dell'oracolo di test, una validazione critica indipendente dal modello generatore e l'estensione del dataset a categorie di difetti diverse rappresentano le direzioni più promettenti per consolidare i risultati qui presentati.

Bibliografia

[1] Seyed Moein Abtahi e Akramul Azim. «Augmenting Large Language Models with Static Code Analysis for Automated Code Quality Improvements». In: ().

[2] Alfred V Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd. Pearson Education, 2006.

[3] Google Cloud Community / Anthropic. *Structuring Agent Instructions with XML for Dramatically Better LLM Performance*. Technical Report / Best Practices for Prompt Engineering. 2026.

[4] Yuntao Bai et al. «Constitutional AI: Harmlessness from AI Feedback». In: *arXiv preprint arXiv:2212.08073* (2022).

[5] Islem Bouzenia, Premkumar T. Devanbu e Michael Pradel. «RepairAgent: An Autonomous, LLM-Based Agent for Program Repair». In: *Proceedings of the 47th International Conference on Software Engineering (ICSE)*. 2025.

[6] Tom Brown et al. «Language models are few-shot learners». In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[7] Max Brunsfeld e Tree-sitter contributors. *Tree-sitter: An Incremental Parsing System for Programming Tools*. 2018. URL: `https://tree-sitter.github.io/tree-sitter/` (visitato il 28/06/2026).

[8] Viola Campos et al. *Exploring Generalizable Automated Program Repair with Large Language Models*. 2025.

[9] Lingjiao Chen, Matei Zaharia e James Zou. «FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance». In: *arXiv preprint arXiv:2305.05176* (2023).

[10] Shehzaad Dhuliawala et al. «Chain-of-Verification Reduces Hallucination in Large Language Models». In: *arXiv preprint arXiv:2309.11495* (2023).

[11] Sena Dikici e Turgay Tugay Bilgin. «Advancements in automated program repair: a comprehensive review». In: *Knowledge and Information Systems* 67 (mar. 2025), pp. 4737–4783.

[12] Yilun Du et al. «Improving Factuality and Reasoning in Language Models through Multiagent Debate». In: *arXiv preprint arXiv:2305.14325* (2023).

[13] Thomas Durieux et al. «Dynamic Patch Generation for Null Pointer Exceptions using Metaprogramming». In: *SANER '17*. 2017, pp. 349–358.

[14] Zuocheng Feng et al. «Concurrency Bug Detection via Static Analysis and Large Language Models». In: *Future Internet* 17 (2025), p. 578. DOI: `10.3390/fi17120578`.

[15] Jie Huang et al. «Large Language Models Cannot Self-Correct Reasoning Yet». In: *The Twelfth International Conference on Learning Representations*. 2024.

[16] Kai Huang et al. «A Survey on Automated Program Repair Techniques». In: *ACM Computing Surveys* 37.4 (ago. 2022).

[17] Kai Huang et al. «Evolving Paradigms in Automated Program Repair: Taxonomy, Challenges, and Opportunities». In: *ACM Computing Surveys* 57.2 (ott. 2024).

[18] Yue Jia e Mark Harman. «An analysis and survey of the development of mutation testing». In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[19] Carlos E Jimenez et al. «SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering». In: *arXiv preprint arXiv:2405.15793* (2024).

[20] René Just, Darioush Jalali e Michael D. Ernst. «Defects4J: A database of existing faults to enable controlled testing studies for Java programs». In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. ACM. 2014, pp. 437–440. DOI: 10.1145/2610384.2628055.

[21] Abdullah Talha Kabakuş. «Evaluating the Impact of Temperature and Instruction Strategies on Hallucination in Large Language Models». In: *Gazi University Journal of Science* (2026).

[22] Jindae Kim e Jaewoo Song. «Identifying Root Causes of Null Pointer Exceptions with Logical Inferences». In: *Manuscript submitted to ACM* (dic. 2024).

[23] Jiaolong Kong, Xiaofei Xie e Shangqing Liu. «Demystifying Memorization in LLM-Based Program Repair via a General Hypothesis Testing Framework». In: *Proc. ACM Softw. Eng.* 2.FSE (lug. 2025). DOI: 10.1145/3729390.

[24] Xuan-Bach D. Le. «Towards Efficient and Effective Automatic Program Repair». In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, set. 2016.

[25] Fengjie Li et al. *Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis*. 2024.

[26] Hunter Lightman et al. «Let's Verify Step by Step». In: *arXiv preprint arXiv:2305.20050* (2023).

[27] Nelson F. Liu et al. «Lost in the middle: How language models use long contexts». In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 177–196.

[28] Long Ouyang et al. «Training language models to follow instructions with human feedback». In: *Advances in neural information processing systems* 35 (2022), pp. 27730–27744.

[29] Anvith Pabba et al. «SemAgent: A Semantics Aware Program Repair Agent». In: (2025).

[30] Arjun Panickssery, Samuel R Bowman e Shi Feng. «LLM Evaluators Recognize and Favor Their Own Generations». In: *arXiv preprint arXiv:2404.13076* (2024).

[31] Matthew M. Papi et al. «Practical pluggable types for Java». In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*. ACM. 2008, pp. 201–212. DOI: 10.1145/1390630.1390656.

[32] Zechen Qi et al. «An analysis of patch plausibility and correctness for generate-and-validate object-oriented program repair». In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM. 2015, pp. 24–36.

[33] Yujia Qin et al. «ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs». In: *arXiv preprint arXiv:2307.16789* (2023).

[34] Melika Sepidband et al. «RGFL: Reasoning Guided Fault Localization for Automated Program Repair Using Large Language Models». In: *Manuscript submitted to ACM* (gen. 2026). URL: https://doi.org/10.1145/nnnnnnn.nnnnnnn.

[35] Mrinank Sharma et al. «Towards Understanding Sycophancy in Language Models». In: *arXiv preprint arXiv:2310.13548* (2023).

[36] Noah Shinn et al. «Reflexion: Language Agents with Verbal Reinforcement Learning». In: *Advances in Neural Information Processing Systems* 36 (2023).

[37] Edward K. Smith et al. «Is the cure worse than the disease? Overfitting in automated program repair». In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM. 2015, pp. 532–543. DOI: 10.1145/2786805.2786825.

[38] [...] Sun et al. «Exploring the Impact of Temperature on Large Language Models: Hot or Cold?» In: *arXiv preprint arXiv:2506.07295* (2024).

[39] Karthik Valmeekam, Matthew Marquez e Subbarao Kambhampati. «Can Large Language Models Really Improve by Self-critiquing Their Own Plans?» In: *arXiv preprint arXiv:2310.08118* (2023).

[40] Autori Vari. «Exploring Generalizable Automated Program Repair with Large Language Models». In: *arXiv preprint arXiv:2506.03283* (2026).

[41] Lei Wang et al. «Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models». In: *arXiv preprint arXiv:2305.04091* (2023).

[42] Jason Wei et al. «Chain-of-Thought Prompting Elicits Reasoning in Large Language Models». In: *Advances in Neural Information Processing Systems (NeurIPS)* 35 (2022), pp. 24824–24837.

[43] Jules White et al. «A prompt pattern catalog to enhance prompt engineering with chatgpt». In: *arXiv preprint arXiv:2302.11382* (2023).

[44] [...] Wu et al. «Chain-of-Thought Prompting Obscures Hallucination Cues in Large Language Models: An Empirical Evaluation». In: *Findings of the Association for Computational Linguistics*. 2025.

[45] Chunqiu Steven Xia, Yifeng Ding e Lingming Zhang. «Revisiting the Plastic Surgery Hypothesis via Large Language Models». In: *arXiv preprint arXiv:2303.10494* (dic. 2024).

[46] Chunqiu Steven Xia e Lingming Zhang. «Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT». In: *arXiv preprint arXiv:2304.00385* (2023).

[47] Xuezheng Xu et al. «VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences». In: IEEE, 2019.

[48] Jinqiu Yang et al. «Better test cases for better automated program repair». In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM. 2017, pp. 831–841. DOI: 10.1145/3106237.3106274.

[49] Shunyu Yao et al. «ReAct: Synergizing Reasoning and Acting in Language Models». In: *International Conference on Learning Representations (ICLR)*. 2023.

[50] Xin Yin et al. «Thinkrepair: Self-directed Automated Program Repair». In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2024, pp. 1274–1286.

[51] Jiayi Zhang et al. «Repair Ingredients Are All You Need: Improving Large Language Model-Based Program Repair via Repair Ingredients Search». In: *Procee-*

dings of the 48th International Conference on Software Engineering (ICSE 2026). Rio de Janeiro, Brazil: ACM, 2026.

[52] Wuyang Zhang et al. «SemanticForge: Repository-Level Code Generation through Semantic Knowledge Graphs and Constraint Satisfaction». In: (2025).

[53] Ziyao Zhang et al. «LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation». In: (2024).

[54] Zihao Zhao et al. «Calibrate before use: Improving few-shot performance on language models». In: *International Conference on Machine Learning (ICML)*. 2021, pp. 12697–12706.

[55] Lianmin Zheng et al. «Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena». In: *Advances in Neural Information Processing Systems* 36 (2023).

[56] [...] Zhu et al. «StructuredRAG: JSON Response Formatting with Large Language Models». In: *arXiv preprint arXiv:2408.11061* (2024).

[57] Taohong Zhu et al. «Specification Vibing for Automated Program Repair». In: *arXiv preprint arXiv:2602.08263* (feb. 2026).

Ringraziamenti

Vorrei ringraziare prima di tutti i miei genitori, Patrizia e Nicola, che mi hanno dato l'opportunità di intraprendere questo percorso di studi e che hanno creduto in me, trasmettendomi fiducia in tutti questi anni e rendendo il viaggio più leggero.

Ringrazio gli amici che ho conosciuto in questi anni di università, per il supporto e l'aiuto reciproco che ci siamo dati nei momenti di difficoltà. Ricordo con molta felicità i momenti passati insieme durante le lezioni, le pause e gli esami: questi anni non sarebbero stati gli stessi senza di voi.

Ai miei amici più cari, Angelo, Giovanni, Pasquale, Raffaele e Tammaro, diventati ormai la costante della mia vita da diversi anni: sono molto orgoglioso della nostra amicizia e voglio dedicarvi questo spazio.

A Manu va un ringraziamento speciale, perché mi hai dato la forza necessaria per completare quest'ultimo anno. Da quando ci siamo conosciuti, il modo in cui mi hai supportato è stato molto prezioso per me, e spero di riuscire a ricambiare questo supporto restando al tuo fianco passo dopo passo. Sono davvero felice di vivere questo momento con te, e sono consapevole che è solo il primo di tanti traguardi che ci aspettano insieme.

Ai miei nonni Pasquale, Maria e Mario, che mi hanno visto crescere e che non possono essere qui in questo momento: vi porto nel mio cuore e vi voglio ringraziare per tutti gli insegnamenti che mi avete dato. Se oggi sono la persona che sono, è grazie a voi.