ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea in Informatica

Web Scraping e LLM:

Confronto di metodologie per l'estrazione di dati non strutturati da fonti eterogenee tramite Large Language Model

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Alessandro Amella

Sessione marzo 2026
Anno Accademico 2024/2025

Web scraping

Extract, Transform, Load

AI

Data extraction

Large Language Model

A Fiocchetto

Abstract

Il web scraping tradizionale, basato su regole sintattiche, è intrinsecamente fragile alle modifiche strutturali dei siti web. I *Large Language Model* (LLM) offrono un cambio di paradigma: descrivere alla macchina «cosa» estrarre anziché «come» trovarlo. L'IA generativa introduce però nuove sfide: costi operativi, latenza e allucinazioni.

Questa dissertazione analizza tali trade-off tramite SWEET (*Semantic Web Extraction & Evaluation Tool*), un framework di benchmark sviluppato ad hoc e testato su un caso reale (avvisi di sciopero). Si valutano accuratezza (F1-score) ed efficienza confrontando modelli (GPT, Gemini, Llama, DeepSeek), schemi di validazione e preprocessing, dalle euristiche classiche ai recenti *Small Language Model* (SLM).

I risultati mostrano che l'uso degli SLM per il preprocessing garantisce un'elevata compressione del documento, ma rischia di omettere metadati rilevanti. Su fonti semi-strutturate a struttura relativamente stabile prevale un approccio ibrido: pre-filtrare il DOM deterministicamente riduce i token in input del 99,5%, mantenendo un F1-score oltre 0,95. Sul fronte della validazione dei dati, alleggerire il carico cognitivo dell'IA richiedendo un output flessibile, normalizzato a valle via codice, riduce le allucinazioni fino al 45%. Infine, test di mutazione del DOM («DOM chaos») confermano l'immunità strutturale degli LLM davanti al collasso degli scraper classici.

L'evidenza sperimentale dimostra che i modelli generativi non costituiscono una soluzione universale, ma impongono un nuovo trade-off architetturale: l'azzeramento del debito tecnico manutentivo avviene al prezzo di maggiori latenze, costi di inferenza e del nuovo rischio delle allucinazioni, che sostituiscono il fallimento esplicito (*fail-fast*) degli scraper tradizionali con errori subdoli e silenziosi. L'uso degli LLM non elimina il software classico, ma trasforma il web scraping da una fragile ingegneria sintattica a una robusta orchestrazione semantica.

Indice

1 Introduzione **1**

2 Stato dell'arte: dal web scraping tradizionale ai Large Language Model **9**

2.1 Web scraping tradizionale: potenza e debolezza 9
2.2 L'avvento degli LLM nell'estrazione di informazioni 11
2.2.1 Parsing semantico diretto con LLM 12
2.2.2 Pipeline ibride con «DOM distillation» 13
2.3 Evoluzione delle strategie di preprocessing 14
2.3.1 Riduzione euristica e strutturale 14
2.3.2 Conversione in formati intermedi 15
2.3.3 Uso di Small Language Model per l'estrazione del contenuto . 15
2.4 Il problema delle allucinazioni 16
2.4.1 Definizione e tassonomia 16
2.4.2 Perché le allucinazioni sono un problema critico nello scraping 17
2.4.3 Tecniche di mitigazione 18
2.5 Output strutturato e carico cognitivo 18
2.6 Il limite della latenza 19
2.7 Valutazione dell'estrazione 20

3 Disegno sperimentale e metodologia di valutazione **23**

3.1 Struttura del dataset e schema di ground truth 23
3.2 Disegno sperimentale e variabili di valutazione 24
3.3 Strategie di preprocessing valutate 25
3.4 Modelli LLM testati 27
3.5 Schema di validazione: *strict* vs *lenient* 28
3.6 Confronto tra istruzioni generiche ed esplicite 28
3.7 Disegno sperimentale per il test di resilienza 29

4 Implementazione di SWEET **31**

4.1 Costruzione e archiviazione del dataset 31
4.2 Architettura di SWEET e design pattern 32
4.2.1 Panoramica del modulo 32
4.2.2 Astrazione delle API con il pattern adapter 35
4.2.3 Pattern strategy per il preprocessing e il parsing 38
4.2.4 La funzione `applyStrategy` per il preprocessing 41
4.3 Orchestrazione e pipeline AI 41
4.3.1 Implementazione degli schemi *strict* e *lenient* 43
4.3.2 Costruzione dinamica del prompt 46
4.3.3 Affidabilità, caching e raccolta dati 46
4.4 Analisi della resilienza e metodologia di valutazione 48
4.4.1 Il generatore di «DOM chaos» 48
4.4.2 Algoritmo di scoring 49
4.4.3 Visualizzazione dei risultati 51

5 Valutazione dei risultati: accuratezza, costi e resilienza strutturale 53
5.1 Metodologia di valutazione dell'accuratezza 53
5.2 L'importanza del prompt engineering 54
5.2.1 Conseguenze di un prompt generico 54
5.2.2 Complessità di estrazione per tipologia di campo 56
5.3 Impatto del preprocessing e costi 59
5.3.1 Compressione per strategie di preprocessing 59
5.3.2 Impatto sull'accuratezza 59
5.3.3 Analisi costo-efficacia 62
5.4 Analisi della latenza e dei tempi di esecuzione 67
5.5 Carico cognitivo nello schema di output 68
5.5.1 Accuratezza complessiva e precisione 68
5.6 Quantificare la resilienza strutturale 70
5.6.1 Osservazione sulla strategia ibrida 72

6 Conclusioni e sviluppi futuri 73

Appendici

Appendice A Prompt di estrazione
A.1 Prompt generico (fase iniziale) . . . .
A.2 Prompt esplicito (fase finale) . . . .

Appendice B Uso di strumenti di intelligenza artificiale generativa

Elenco delle tabelle

3.1 Modelli LLM e tariffe API 27

4.1 Stack tecnologico e librerie principali 34
4.2 Logica di scoring 51

5.1 Distribuzione degli errori per campo 58
5.2 Performance dei modelli e strategie di preprocessing 65
5.3 Costi ed errori per modello e strategia 66

Elenco delle figure

4.1 Architettura di SWEET . . . 33
4.2 Diagramma UML del pattern adapter (integrazione modelli IA) . . . 36
4.3 Diagramma UML del pattern strategy (preprocessing e parsing) . . . 39

5.1 Phantom strike con prompt generico . . . 55
5.2 F1-score medio per modello . . . 57
5.3 Compressione per strategie di preprocessing . . . 60
5.4 F1-score medio per strategia . . . 61
5.5 Costi vs accuratezza per strategie conservative . . . 63
5.6 Costi vs accuratezza per strategie più aggressive . . . 64
5.7 Latenza media per modello e strategia . . . 67
5.8 F1-score schema strict vs lenient . . . 69
5.9 Tasso di allucinazione schema strict vs lenient . . . 70
5.10 F1-score DOM intatto vs alterato . . . 71

Capitolo 1

Introduzione

Nell'era digitale contemporanea, i dati rappresentano una risorsa di inestimabile valore per l'analisi, la ricerca e l'innovazione. Il World Wide Web costituisce l'archivio di informazioni più vasto e in continua espansione. Tuttavia, la stragrande maggioranza di queste informazioni è pubblicata in formati pensati per la fruizione umana, come pagine HTML, e non per essere elaborata automaticamente dalle macchine. È in questo contesto che si inserisce il **web scraping**, una tecnica per l'estrazione automatica di dati (*data scraping*) da siti web.

Il web scraping si differenzia sostanzialmente dall'utilizzo di API. Queste ultime sono interfacce progettate specificamente per permettere a programmi e applicazioni di comunicare e scambiare dati in modo strutturato e predefinito, mentre il web scraping interviene laddove tali interfacce non esistono o non sono sufficienti. Il suo scopo è interpretare la struttura di una pagina web pensata per un utente umano, estrarne le informazioni rilevanti e trasformarle in un formato strutturato e facilmente utilizzabile da sistemi automatizzati, come CSV, JSON o all'interno di un database. In ambito di *data engineering*, questo flusso si configura come una vera e propria pipeline ETL (Extract, Transform, Load), in cui lo scraping costituisce la fase di estrazione (*Extract*), propedeutica alla successiva pulizia (*Transform*) e archiviazione (*Load*) del dato.

Le applicazioni di questa tecnica sono estremamente vaste e trasversali: monitoraggio dei prezzi della concorrenza, analisi di mercato, ricerca scientifica, aggregatori di contenuti e, più recentemente, raccolta di dati per l'addestramento di modelli di IA [KS25]. Con l'avvento dei modelli linguistici di grandi dimensioni (LLM), il web scraping è diventato un processo indispensabile per la costruzione dei vasti dataset testuali necessari al loro addestramento [BMR+20].

Tuttavia, trasformare il web da un ambiente eterogeneo e intrinsecamente mutevole in una fonte di dati strutturata e affidabile è un'operazione che comporta sfide ingegneristiche e manutentive significative. Implementare una pipeline di acquisizione non è sufficiente: la difficoltà principale è garantirne il funzionamento nel tempo. Per comprendere l'origine di questi problemi è utile dividere il processo di web scraping in due fasi con natura e criticità diverse: il *crawling* (il recupero fisico della pagina web, la gestione di proxy, il superamento di barriere di rete e l'eventuale rendering di codice JavaScript) e l'estrazione o *parsing* (l'identificazione e la strutturazione semantica dei dati all'interno del documento sorgente).

Questa dissertazione si concentra esclusivamente sulla seconda fase: assumendo di aver già delegato a strumenti terzi il recupero del codice sorgente, l'obiettivo è determinare come estrarne le informazioni in modo accurato e resiliente.

Concentrandosi sulla sola estrazione, il problema centrale che questo lavoro intende affrontare è la **fragilità strutturale** degli approcci tradizionali. Fin dagli albori del web, l'estrazione dati si è basata su regole deterministiche fortemente accoppiate alla struttura sintattica della pagina di destinazione (percorsi gerarchici, identificativi di stile o pattern testuali). Questo paradigma fa sì che qualsiasi modifica, anche minima e puramente visuale, al layout del sito possa «rompere» la logica di parsing, rendendo il sistema incapace di localizzare i dati correttamente. A ciò si aggiunge una barriera difensiva da parte dei proprietari dei siti, che spesso offuscano intenzionalmente il codice per ostacolare l'estrazione automatica. Il risultato è un forte *debito tecnico*: i tassi di rottura mensili oscillano tra il 5% e il 20% su grandi portfolio di domini, rendendo necessario un intervento umano costante per il ripristino delle funzionalità [Kul26].

Di fronte a questa cronica fragilità strutturale, l'avvento dei Large Language Model (LLM) rende praticabile un cambiamento di paradigma: invece di specificare *come* recuperare un'informazione, è ora possibile comunicare al sistema *cosa* si desidera ottenere, espresso in linguaggio naturale. La responsabilità di interpretare la struttura della pagina, di individuare l'informazione richiesta e di estrarla nel formato corretto viene delegata al modello, che riceve in input il documento sorgente e una descrizione dell'obiettivo di estrazione.

Questo cambio di paradigma è reso possibile dalle capacità di comprensione semantica dei moderni LLM, che hanno dimostrato di poter eseguire task complessi in modalità *few-shot* (in cui vengono forniti alcuni esempi nel prompt) o *zero-shot* (senza alcun esempio, sulla base della sola istruzione), ricevendo istruzioni esclusivamente in linguaggio naturale [BMR+20]. Applicato al web scraping, ciò significa che è possibile descrivere cosa estrarre senza dover conoscere né specificare la struttura HTML sottostante. Studi recenti confermano la fattibilità di questo ap-

proccio, mostrando come gli LLM possano essere impiegati per estrarre dati strutturati da pagine web con ottimi risultati, a patto di utilizzare formati di input appropriati [KKJ25, AW24].

L'adozione dei Large Language Model sposta il problema dalla fragilità sintattica all'**affidabilità semantica**. Se i parser deterministici garantiscono l'esattezza (o il fallimento esplicito), i modelli generativi introducono variabili legate a costi, latenza e coerenza del dato, che richiedono una valutazione più attenta.

In primo luogo, ogni chiamata a un LLM ha un costo economico non trascurabile, che può diventare significativo quando il numero di pagine da elaborare è elevato. La dimensione dell'input influisce direttamente sul consumo di token e quindi sulla spesa: inviare al modello l'intero codice HTML di una pagina, oltre a introdurre una quantità di rumore di fondo tale da poter confondere il modello, comporta costi molto maggiori rispetto all'utilizzo di rappresentazioni più compatte[1]. Riduzioni anche di un ordine di grandezza nel numero di token in input possono quindi tradursi in risparmi economici rilevanti, come verrà mostrato nel capitolo 5.

Un secondo aspetto riguarda l'affidabilità del risultato. A differenza degli strumenti di scraping tradizionali, che in caso di errore tendono a fallire in modo evidente, i modelli generativi possono produrre output formalmente corretti ma semanticamente errati, introducendo informazioni non presenti nella sorgente o interpretazioni scorrette del contenuto [BPK+25]. Questo fenomeno, noto come *allucinazione* (approfondito nel capitolo 2.4), è particolarmente insidioso perché la plausibilità superficiale del testo generato rende difficile distinguere automaticamente i dati validi da quelli «inventati».

Infine, l'utilizzo di LLM introduce anche un costo computazionale in termini di tempo di elaborazione, generalmente superiore rispetto all'esecuzione di codice deterministico, che può influire sulla progettazione di sistemi che devono operare su larga scala o con vincoli di tempo stringenti.

L'obiettivo di questa dissertazione è analizzare questo nuovo equilibrio. Nello specifico, si intende indagare direttamente come utilizzare al meglio gli LLM per lo scraping, cercando il compromesso ideale tra i benefici della resilienza semantica e le nuove problematiche introdotte.

A partire da questa cornice, la tesi si propone di rispondere alle seguenti domande di ricerca:

[1] A titolo indicativo, a marzo 2026 modelli come Gemini 3.1 Pro possono costare fino a 18 dollari per milione di token in output, contro gli 0,42 dollari di alternative come DeepSeek-V3.2. Per i modelli utilizzati in questa tesi e i relativi costi, si rimanda alla tabella 3.1.

1. **Che impatto hanno le diverse strategie di preprocessing sul consumo di token e sull'accuratezza dei dati estratti?** Verranno confrontate tecniche come la rimozione di elementi boilerplate, la conversione in Markdown, l'uso di modelli piccoli specializzati e l'approccio ibrido di «distillazione» del DOM, valutando non solo l'accuratezza dell'estrazione ma anche il costo in termini di token e latenza.
2. **In che misura la scelta del modello LLM e del formato di output influenza la qualità dei dati estratti?** Verranno messi a confronto modelli di diverse dimensioni e costi (GPT-5, Gemini, Llama, DeepSeek) e due filosofie di validazione: uno schema di output rigido, che richiede al modello di aderire perfettamente a formati prestabiliti, e uno schema «tollerante», che delega la normalizzazione finale al codice deterministico.
3. **È possibile quantificare il trade-off tra resilienza strutturale (capacità di adattarsi a cambiamenti del sito) e costo computazionale/manutentivo?** Attraverso un test di mutazione del DOM («DOM chaos»), si misurerà la degradazione delle performance di uno scraper tradizionale rispetto a uno basato su LLM a fronte di modifiche casuali della struttura HTML.

Il dominio prescelto come caso di studio è quello degli avvisi di sciopero nel settore dei trasporti pubblici italiani. Si tratta di un ambito particolarmente adatto alla sperimentazione per l'eterogeneità delle fonti: alcune, come i siti web di Trenord o EAV, presentano sezioni con una struttura HTML stabile e prevedibile, dedicata quasi esclusivamente agli annunci di sciopero; altre, come il sito di ATAC, pubblicano sotto la medesima categoria annunci di nuovi scioperi alternati a comunicati di natura diversa (revoche, report di adesione, aggiornamenti di servizio), il che impone al sistema una capacità di classificazione semantica oltre alla mera estrazione; altre ancora, come Trenitalia TPER, pubblicano gli avvisi esclusivamente in formato PDF, in cui le informazioni sono presentate in puro linguaggio naturale, senza alcuna struttura semantica esplicita.

Questa eterogeneità rende il dominio ideale per valutare le diverse strategie di estrazione su più dimensioni: accuratezza sul dato estratto, capacità di discriminare contenuti rilevanti dal rumore della pagina, robustezza a variazioni strutturali, e infine costo computazionale e latenza. L'obiettivo non è stabilire una gerarchia assoluta tra gli approcci, ma caratterizzarne i trade-off in modo empirico, così da fornire indicazioni utili alla scelta della strategia più adatta al contesto.

Per rispondere a questi interrogativi attraverso un'indagine empirica, il contributo ingegneristico di questo lavoro consiste nell'analisi dei trade-off tra diverse strategie di estrazione basate su LLM, variando sistematicamente tecniche di preprocessing,

modelli e formulazione dei prompt. I risultati mostrano che non esiste una soluzione universalmente superiore: anche all'interno dell'approccio basato su LLM, la scelta ottimale dipende dal contesto applicativo e dai vincoli del sistema, bilanciando accuratezza, costo computazionale e robustezza a variazioni strutturali.

Per rispondere alle domande di ricerca, è stata condotta una campagna sperimentale su un dataset di centinaia di avvisi di sciopero, composto esclusivamente da avvisi storici reali. Sono state valutate diverse combinazioni di strategie di preprocessing, modelli LLM e schemi di output, misurando per ciascuna:

- L'**accuratezza** dell'estrazione, misurata tramite l'*F1-score*. La ragione di questa scelta è spiegata più approfonditamente nella sezione 5.1.
- Il **costo** per avviso, stimato in base al consumo di token e alle tariffe delle API.
- La **latenza** media.
- Il **tasso di allucinazione**, con particolare attenzione ai falsi positivi.

Per rispondere a queste domande di ricerca, la presente dissertazione propone un'analisi empirica supportata da **SWEET** (*Semantic Web Extraction & Evaluation Tool*), un framework di valutazione sperimentale sviluppato appositamente per eseguire migliaia di estrazioni controllate, isolando le variabili in gioco e misurandone gli impatti in modo quantitativo.

Al fine di fornire fin da subito una visione d'insieme dell'intero lavoro di tesi, di seguito viene presentata una sintesi dell'architettura metodologica adottata, delle scelte implementative, dei risultati più rilevanti emersi dalla sperimentazione e delle conclusioni tratte.

Il contesto tecnologico e le problematiche attuali Il **capitolo 2** contestualizza il problema partendo dalle fondamenta del web scraping tradizionale. Si analizza come l'approccio classico, basato sul *knowledge engineering* (selettori CSS, XPath, espressioni regolari), offra un'esecuzione istantanea e quasi a costo zero, ma sconti un inesorabile debito tecnico dovuto alla sua fragilità strutturale: ogni minima variazione del DOM rende lo scraper inutilizzabile. L'introduzione dei Large Language Model risolve questo problema spostando l'asse dall'estrazione *sintattica* a quella *semantica*. Tuttavia, la letteratura recente evidenzia come questa transizione introduca nuove criticità: il rischio di allucinazioni (intrinseche ed estrinseche), i limiti legati al carico cognitivo richiesto per generare output JSON complessi e i costi operativi derivanti dall'elaborazione di interi documenti HTML. Il capitolo traccia quindi l'evoluzione delle tecniche di mitigazione, dall'uso di parser ibridi (*DOM di-*

stillation) all'impiego di Small Language Model (SLM) per la pulizia preliminare del contenuto.

Metodologia e disegno sperimentale Per validare le ipotesi, il **capitolo 3** definisce il protocollo sperimentale. Il dominio scelto come caso di studio è quello degli avvisi di sciopero nel settore del trasporto pubblico italiano (analizzando fonti come Trenord, ATAC, EAV e Trenitalia TPER). La natura eterogenea di questi documenti – che variano da pagine web altamente strutturate a comunicati in puro testo libero all'interno di file PDF – fornisce un banco di prova ideale. Il disegno sperimentale incrocia diverse variabili: quattro modelli di IA di diverse fasce di prezzo e architettura (GPT-5 nano, Gemini 3.1 Flash Lite, Llama-4 Scout 17B, DeepSeek-V3.2), sei strategie di preprocessing (da una semplice pulizia dell'HTML basata su euristiche, all'uso di Small Language Model specializzati come Jina Reader e MinerU, fino a tecniche ibride) e due approcci di validazione dello schema dati («strict» contro «lenient»). La valutazione si fonda su metriche derivate dall'NLP: precision, recall e F1-score, calcolate granularmente sui campi estratti. Completano il quadro valutativo le misurazioni di latenza e consumo di token.

Implementazione di SWEET Il **capitolo 4** illustra l'architettura software di **SWEET**. Sviluppato in TypeScript con il framework NestJS, il cuore logico del sistema risiede nel modulo `BenchmarksModule`. Per gestire la complessità e frammentazione delle chiamate ai vari provider di intelligenza artificiale, il sistema è stato ingegnerizzato facendo largo uso di design pattern: un pattern *adapter* astrae le comunicazioni con le API (OpenAI, Google, Groq, DeepSeek), mentre un pattern *strategy* orchestra i diversi parser (manuali, basati su AI o ibridi). Il motore di esecuzione integra meccanismi per la sperimentazione su larga scala: controllo della concorrenza per non superare i *rate limit* imposti dai provider delle API, un sistema di caching basato su hash MD5 per evitare costi ridondanti e garantire la riproducibilità, e algoritmi di normalizzazione «lenient» per trasformare le estrazioni semantiche grezze in dati strutturati. Viene inoltre descritto lo strumento di «DOM chaos», un algoritmo sviluppato appositamente per mutare l'albero HTML alterandone nomi di classi e tag ma preservandone l'aspetto visivo, in modo da poter testare la resilienza strutturale dei diversi approcci.

Risultati: accuratezza, efficienza economica e resilienza Il **capitolo 5** presenta e discute i risultati ottenuti dalla campagna sperimentale, fornendo risposte quantitative ai quesiti di ricerca iniziali. I dati raccolti attraverso le diverse configurazioni del framework evidenziano alcuni aspetti determinanti per la progettazione di sistemi di estrazione moderni:

- L'indagine sulla formulazione delle istruzioni mostra come l'impiego di **prompt generici** porti a errori sistematici nella classificazione di documenti ambigui, quali revoche o differimenti, che vengono scambiati per nuovi annunci in circa la metà dei casi. L'adozione di direttive esplicite e la fornitura di esempi concreti (*few-shot prompting*) permettono di stabilizzare le prestazioni, portando anche i modelli più economici, come GPT-5 nano e DeepSeek-Chat, a superare un F1-score di 0,95.
- L'analisi delle **strategie di preprocessing** indica che l'uso di documenti grezzi è economicamente oneroso, mentre una compressione troppo spinta tramite modelli linguistici di piccole dimensioni rischia di rimuovere metadati necessari alla corretta interpretazione del contesto. L'approccio che combina la distillazione mirata del DOM alla conversione in Markdown si è dimostrato il più equilibrato, riducendo il volume dei token del 99,5% senza compromettere la precisione dell'estrazione finale.
- Il confronto tra gli **schemi di validazione** suggerisce che alleggerire il carico cognitivo dei modelli, richiedendo un output meno vincolato (schema *lenient*) e affidando la normalizzazione finale a script deterministici, riduca sensibilmente la propensione all'allucinazione. Questo accorgimento si è rivelato efficace soprattutto per i modelli *open-weight*, con riduzioni dell'errore nell'ordine del 30-40% in alcuni casi.
- I test condotti in condizioni di **alterazione strutturale** (*DOM chaos*) confermano la fragilità dei parser tradizionali, la cui accuratezza scende drasticamente a fronte di modifiche sintattiche che non alterano il contenuto visibile. Al contrario, le pipeline basate su modelli linguistici mantengono prestazioni costanti, dimostrando un'immunità strutturale che compensa i tempi di risposta superiori, i quali oscillano tra pochi secondi e circa quindici secondi a seconda dell'hardware e del modello impiegato.

Conclusioni e prospettive Infine, il **capitolo 6** riassume le evidenze emerse e analizza i criteri di scelta tra le diverse metodologie di estrazione. L'indagine evidenzia come la configurazione ottimale di una pipeline di scraping non dipenda da un unico parametro, ma derivi da un bilanciamento tra accuratezza del dato, costi di inferenza, latenza di risposta e stabilità strutturale delle fonti. Vengono discusse le diverse modalità di integrazione dei modelli linguistici, valutando l'efficacia di approcci completamente automatizzati rispetto a soluzioni ibride che mantengono componenti deterministiche per il filtraggio e la validazione. In chiusura, si discutono alcune direzioni per sviluppi futuri, quali l'automazione dei meccanismi di «self-healing» per i parser e l'uso di modelli multimodali che interpretano il rendering visivo delle pagine web senza dipendere dal codice sorgente.

Capitolo 2

Stato dell'arte: dal web scraping tradizionale ai Large Language Model

Prima di analizzare l'integrazione dei Large Language Model nel web scraping, questo capitolo contestualizza il problema: vengono dapprima esaminate le tecniche tradizionali, inquadrandole formalmente e descrivendone potenzialità e limiti strutturali, per poi passare allo stato dell'arte nell'uso degli LLM per l'estrazione di informazioni, alle strategie di preprocessing e al problema delle allucinazioni.

2.1 Web scraping tradizionale: potenza e debolezza

Il web scraping, definito in senso lato come l'insieme di tecniche per l'estrazione automatica di dati da siti web, è da decenni uno strumento indispensabile per ricercatori, analisti e aziende per trasformare documenti pensati per la lettura umana in dati strutturati.

Queste tecniche di estrazione rientrano nel dominio più ampio dell'Information Extraction (IE). Già in lavori come quello di Flesca et al. [FMM+04], le tecniche di IE applicate al web mirano a trasformare informazioni testuali in formati strutturati mediante l'uso di regole di estrazione. Un insieme di regole progettato per identificare e assegnare significato semantico ai dati all'interno di uno specifico sito web prende il nome di «wrapper».

Per comprendere la natura dei wrapper, Flesca et al. propongono una tassonomia delle pagine web basata sul loro grado di strutturazione:

- **Pagine strutturate**: dove le informazioni sulla struttura sono esplicite e i dati possono essere estratti unicamente attraverso vincoli sintattici e di formattazione.
- **Pagine semi-strutturate**: come la maggior parte dei documenti HTML, in cui non esiste una descrizione separata per il tipo di dato, ma le informazioni e i tag di formattazione coesistono in miscele eterogenee.
- **Pagine non strutturate**: composte prevalentemente da testo libero, dove l'estrazione richiede l'interpretazione semantica del contenuto, spesso utilizzando tecniche di Natural Language Processing (NLP) più che regole sintattiche.

Storicamente, lo sviluppo di wrapper per pagine semi-strutturate (il «web scraping» in senso stretto) si è basato su un paradigma che la letteratura definisce di «knowledge engineering» [FMM+04]. In questo approccio, lo sviluppatore ispeziona manualmente la struttura sorgente del documento e istruisce il computer su *come* trovare esattamente l'informazione desiderata.

Nel corso degli anni, per implementare questi wrapper manuali, sono state sviluppate librerie e framework sempre più avanzati. Un pioniere in questo campo è Beautiful Soup, una libreria Python che permette di navigare l'albero del DOM e ricercare elementi specifici tramite selettori CSS o XPath. Per le estrazioni più mirate da testo semi-strutturato si è fatto largo uso di espressioni regolari (*regex*). Con l'evoluzione verso applicazioni web dinamiche (le *Single Page Application*), sono emersi strumenti come Scrapy e browser automatizzati come Selenium, Puppeteer e Playwright, che permettono di interagire con pagine web come farebbe un utente, eseguendo JavaScript e navigando tra le pagine.

La potenza di questi wrapper tradizionali è innegabile: sono estremamente veloci, precisi e, una volta programmati, hanno un costo di esecuzione computazionale prossimo allo zero. Per un sito con una struttura stabile, un wrapper ben progettato può estrarre dati con un'accuratezza quasi perfetta.

Tuttavia, questa potenza si fonda su una debolezza critica: la fragilità intrinseca. I wrapper tradizionali generati manualmente sono strettamente accoppiati alla struttura sintattica (HTML) della pagina web (o ai vincoli di formattazione di un PDF). Qualsiasi modifica, anche minima e puramente visuale – un aggiornamento del template, il cambio di una classe CSS, una ristrutturazione del layout – può invalidare le regole del wrapper, che non sarà più in grado di localizzare i dati [BDW26, Kul26, FMM+04].

Il mantenimento di questi scraper diventa quindi un'attività onerosa e costante, una forma di *debito tecnico* che si accumula nel tempo. Un caso emblematico, e particolarmente vicino al dominio di questa tesi, è quello del progetto open-source

KDE Itinerary[1]. Nel 2019, Trenitalia modificò il layout dei propri biglietti PDF: orari di partenza e arrivo, prima presenti sulla stessa riga del testo estratto, vennero separati dal numero della carrozza. La regex utilizzata da KDE Itinerary per il parsing, progettata per catturare entrambi gli orari in un'unica corrispondenza, smise di funzionare. Fu necessario riscrivere la logica di estrazione, suddividendo la ricerca in due passaggi sequenziali con due nuove espressioni regolari [noa19]. Questo esempio illustra bene come anche una modifica apparentemente marginale al formato di un documento – impercettibile all'utente finale – possa invalidare completamente la logica di parsing basata su pattern testuali fissi.

2.2 L'avvento degli LLM nell'estrazione di informazioni

In questo contesto di fragilità strutturale ed elevato costo di manutenzione, emergono approcci basati sulla **comprensione semantica** del contenuto anziché sulla sua sola forma sintattica. A differenza del paradigma del *knowledge engineering* discusso nella sezione 2.1, che richiede la definizione di rigidi vincoli strutturali – come selettori CSS, XPath o espressioni regolari – l'integrazione degli LLM permette di individuare le informazioni tramite descrizioni in linguaggio naturale. Il modello assume la responsabilità di interpretare il contesto e localizzare il dato richiesto, operando in modo agnostico rispetto alla specifica gerarchia del DOM o alla formattazione del documento sorgente.

A differenza dei wrapper basati su selettori CSS o espressioni regolari, che cercano pattern sintattici specifici nel codice sorgente, gli LLM sono in grado di «comprendere» il significato semantico del contenuto. Ad esempio, un modello linguistico può riconoscere che «partenza ore 14:30» e «il treno parte alle 14:30» esprimono la stessa informazione, anche se rappresentate con strutture testuali diverse. È proprio questa capacità di generalizzazione semantica a rendere gli LLM naturalmente resilienti ai cambiamenti strutturali che affliggono i wrapper tradizionali: una modifica al layout HTML o al template di un PDF non invalida un sistema di estrazione basato sulla comprensione del contenuto piuttosto che sulla sua forma sintattica.

Questa flessibilità, tuttavia, non significa potersi affidare a istruzioni generiche. In assenza di regole specifiche, i modelli tendono ad applicare euristiche fallaci basate sulla frequenza relativa dei termini nel corpus di addestramento o sulla presenza di parole chiave, usate come «indici» per recuperare nozioni memorizzate anziché per applicare un vero ragionamento logico [MLC+23]. Come verrà dimostrato sperimen-

[1]KDE Itinerary è un'applicazione per la gestione di viaggi che integra biglietti, prenotazioni e orari. Sito ufficiale: `https://apps.kde.org/itinerary/`

talmente nel capitolo 5, un prompt ingenuo (in logica *zero-shot*) porta frequentemente i modelli a estrarre falsi positivi, scambiando ad esempio un report su un evento passato per un annuncio futuro. La complessa logica di dominio, tradizionalmente codificata in euristiche software, non scompare ma deve essere trasferita nel prompt testuale, ricorrendo a regole esplicite ed esempi concreti (*few-shot prompting*) per mitigare questi bias e guidare l'inferenza del modello [MLC+23]. L'efficacia dell'estrazione viene così a dipendere in larga parte dalla qualità dell'istruzione fornita al modello (il *prompt*) – aspetto approfondito nel capitolo 3.6.

Come evidenziato nelle ricerche recenti [AW24, DDL+24], l'applicazione diretta degli LLM all'estrazione dati non è banale. È possibile ricondurre gli approcci attuali a due macro-paradigmi principali attraverso cui i modelli generativi vengono attualmente integrati nelle pipeline di estrazione dati: l'approccio totalmente automatico e le pipeline ibride che combinano tecniche di web scraping tradizionale con l'estrazione semantica guidata da LLM.

2.2.1 Parsing semantico diretto con LLM

Il primo approccio prevede l'uso dell'LLM come motore di estrazione sull'intero documento. In questo paradigma, il contenuto della pagina web (opportunamente ripulito tramite algoritmi automatici di preprocessing) viene passato interamente nel prompt del modello, insieme alle istruzioni su quali dati estrarre e al formato di output desiderato (tipicamente JSON).

Grazie alle ampie *context window* dei modelli moderni, l'LLM analizza il documento ed estrae i dati richiesti senza necessitare di regole specifiche per il singolo sito web [BMR+20, KKJ25]. Questo approccio massimizza la resilienza ai cambiamenti strutturali: finché l'informazione è presente testualmente o logicamente nella pagina, il modello è teoricamente in grado di individuarla, rendendo il sistema quasi del tutto agnostico rispetto al layout HTML [Kul26].

Tuttavia, delegare l'intero onere cognitivo all'LLM presenta svantaggi significativi. In primo luogo, inviare l'intera pagina web al modello, oltre a comportare un elevato consumo di token e costi proporzionali, espone l'LLM a una quantità significativa di rumore (pubblicità, menu, link) che può compromettere l'accuratezza dell'estrazione e causare allucinazioni. È quindi necessario ricorrere a tecniche di preprocessing automatico, ma la loro applicazione richiede un attento bilanciamento: una pulizia troppo aggressiva rischia di rimuovere informazioni semanticamente rilevanti, mentre una troppo conservativa lascia intatto il rumore di fondo. Questo trade-off è analizzato nella sezione 2.3.

2.2.2 Pipeline ibride con «DOM distillation»

Mentre i primi approcci esploravano l'uso degli LLM come estrattori diretti sull'intero documento web, la letteratura più recente evidenzia la necessità di architetture ibride che scompongano il problema in fasi distinte. Un contributo rilevante in questa direzione è il lavoro di Kaur [Kau24], che propone un sistema per l'estrazione di decine di campi complessi da prospetti finanziari in formato HTML.

Anziché delegare l'intero compito a un singolo modello generativo, l'approccio di Kaur utilizza tecniche tradizionali (come Beautiful Soup) per il parsing strutturale iniziale, seguite dall'impiego di modelli NLP specializzati (come RoBERTa) per classificare e filtrare esclusivamente i paragrafi rilevanti. Solo in quest'ultima fase il testo filtrato viene passato a un LLM (ChatGPT) per l'estrazione finale del dato e l'assegnazione del valore.

Questo paradigma, denominato nel presente lavoro «DOM distillation», cambia il ruolo del modello: il codice tradizionale si occupa dell'**estrazione**, mentre l'LLM si concentra sull'**interpretazione** (*parsing*) del testo. È importante notare che, a differenza dei wrapper tradizionali, in questo approccio il codice non è più responsabile di localizzare esattamente il dato, ma solo di identificare un'area semantica più ampia (ad esempio, un paragrafo o una sezione) che contiene l'informazione desiderata. L'LLM, grazie alla sua comprensione semantica, è in grado di estrarre il dato corretto anche se la sua posizione all'interno del testo varia o se è rappresentato in modi diversi.

Se da un lato questo approccio reintroduce una parziale fragilità strutturale (se il sito cambia il layout, la pipeline fallisce), dall'altro riduce significativamente il numero di token in input: non bisogna affidarsi alla capacità della strategia di preprocessing di eliminare il rumore, ma si può semplicemente scartare tutto ciò che non è rilevante, lasciando al modello solo il contenuto pertinente.

A questo vantaggio si aggiunge un'ulteriore prospettiva di automazione per la fase iniziale di knowledge engineering necessaria a individuare l'area semanticamente importante: convertendo il DOM in formati strutturati specifici, è infatti possibile delegare all'LLM l'identificazione dei percorsi corretti [KKJ25]. In questo modo, un programma può ricavare automaticamente i selettori (XPath) esatti associati ai dati di interesse, eliminando la necessità di configurarli a mano e creando una sorta di wrapper «self-healing».

Come verrà illustrato nella sezione 3.3, questa stessa filosofia di decomposizione del task e pre-filtraggio è stata adottata e sperimentata in questo lavoro.

2.3 Evoluzione delle strategie di preprocessing

L'integrazione degli LLM nelle pipeline di web scraping ha ridefinito il ruolo della pre-elaborazione dei documenti. Con l'avvento di modelli dotati di *context window* nell'ordine delle centinaia di migliaia o milioni di token, il limite architetturale legato alla mera quantità di testo in input è stato ampiamente superato. Tuttavia, la capacità di processare interi documenti HTML grezzi non rende il preprocessing obsoleto; al contrario, lo sposta su un piano qualitativo incentrato sulla densità dell'informazione [WSW+25].

L'immissione di un DOM non filtrato presenta infatti due criticità maggiori: da un lato, costi economici di inferenza inutilmente elevati a causa della verbosità sintattica del codice web; dall'altro, un aumento del carico cognitivo per il modello, che deve isolare i dati rilevanti perché sommersi da un grande volume di rumore (script, stili, tag non semantici).

Il preprocessing moderno consiste quindi nella ricerca di un delicato equilibrio: massimizzare il rapporto segnale-rumore senza incorrere in un filtraggio eccessivamente aggressivo. Come emergerà dalle valutazioni nel capitolo 5, una pulizia troppo radicale rischia di eliminare dettagli contestuali cruciali (come note a piè di pagina, elementi strutturali laterali o meta-tag temporali) trasformando un potenziale problema di rumore in un'omissione del dato.

2.3.1 Riduzione euristica e strutturale

Un primo approccio, puramente deterministico, mira a ripulire l'albero del DOM preservandone la struttura originale. Tecniche come lo *slimmed HTML* [KKJ25] consistono nell'eliminare a priori i nodi non testuali (come `<script>`, `<style>`, `<noscript>`) e nel rimuovere gli attributi dei tag (classi CSS, ID e stili inline) che risultano verbosi e semanticamente ridondanti per un modello linguistico.

Questa famiglia di tecniche offre una pulizia conservativa e computazionalmente economica, prevenendo largamente la perdita accidentale di dati (seppur con il rischio minimo di rimuovere attributi semanticamente rilevanti, come date o identificatori numerici). Tuttavia, il tasso di compressione ottenibile è spesso insufficiente per pagine web complesse, lasciando inalterata la verbosità intrinseca dei tag HTML. Una variante di questo approccio, denominata «basic-cleanup», è stata sperimentata nel presente lavoro e valutata nel capitolo 5.

2.3.2 Conversione in formati intermedi

Per superare la verbosità dell'HTML, la ricerca ha esplorato la conversione del documento in formati intermedi più densi a livello semantico. La conversione in Markdown è emersa come uno standard *de facto* in molte pipeline [WSW+25]. Sostituendo la complessa gerarchia di tag con una sintassi testuale leggera (ad esempio, convertendo i tag `<h1>` in `#` e le liste in elenchi puntati), si ottiene una notevole riduzione dei token preservando al contempo l'organizzazione visiva e logica del testo. Bisogna tuttavia considerare che elementi come tabelle o layout complessi possono essere «appiattiti» in modo che la loro rappresentazione testuale perda parte della semantica originale, con potenziali implicazioni per l'accuratezza dell'estrazione.

2.3.3 Uso di Small Language Model per l'estrazione del contenuto

Un'evoluzione più recente delega il compito del preprocessing a modelli linguistici di piccole dimensioni (SLM), addestrati specificamente per estrarre il contenuto principale (*main content*) dal rumore di fondo.

Appartiene a questa categoria **ReaderLM-v2**, un modello da 1,5 miliardi di parametri sviluppato da Jina AI e ottimizzato per convertire HTML eterogeneo in Markdown pulito. L'obiettivo di questo approccio non è la mera compressione spaziale, ma la generazione di un formato intrinsecamente «LLM-friendly»: imponendo una rigorosa coerenza strutturale e risolvendo le ambiguità sintattiche del DOM originale, il modello produce un Markdown ottimizzato che – secondo gli autori – riduce significativamente il carico cognitivo per i modelli generativi impiegati a valle nelle fasi di estrazione [WSW+25].

Un approccio complementare è quello proposto dal framework MinerU-HTML, con il modello **Dripper** [LPN+26]. Invece di tradurre il testo generativamente, Dripper riformula l'estrazione come un task di *sequence labeling*: il modello (da soli 0,6 miliardi di parametri) crea una versione semplificata del DOM in cui gli elementi sono raggruppati in blocchi, classificando ciascun blocco come «contenuto» o «rumore» (boilerplate). Una volta ottenute le etichette, il sistema ricostruisce deterministicamente l'HTML pulito.

Entrambi gli approcci, Jina AI Reader e Dripper, sono stati integrati nelle pipeline sperimentali di questa tesi; i risultati comparativi sono riportati nel capitolo 5.

2.4 Il problema delle allucinazioni

L'integrazione di LLM nelle pipeline di web scraping introduce un nuovo e insidioso rischio: le allucinazioni. Se uno scraper tradizionale, basato su selettori CSS o espressioni regolari, in caso di fallimento restituisce un errore o un risultato vuoto, un LLM può invece generare output che, pur apparendo formalmente corretti e coerenti con il contesto, contengono informazioni non verificate o del tutto inventate. Questo fenomeno, noto in letteratura come *allucinazione*, rappresenta una delle sfide più critiche per l'affidabilità dei sistemi basati su modelli generativi [YLZ+23].

2.4.1 Definizione e tassonomia

Nel contesto della generazione di testo e dell'estrazione di informazioni, l'allucinazione è definita come «contenuto generato che risulta insensato o infedele rispetto al contenuto della fonte fornita in input» [JLF+23]. Applicato al web scraping semantico, questo fenomeno si manifesta quando il Large Language Model produce un output JSON (o dati strutturati) che non riflette in modo accurato o veritiero il documento HTML o Markdown passato nel prompt.

Secondo la classificazione standard proposta nella revisione sistematica di Ji et al. [JLF+23], le allucinazioni si dividono in due macro-categorie principali, entrambe altamente critiche per l'affidabilità di una pipeline di estrazione dati, discusse in dettaglio di seguito.

- **Allucinazione intrinseca (*intrinsic hallucination*)**: l'output generato contraddice esplicitamente le informazioni presenti nella fonte. Nel contesto del nostro dominio applicativo (avvisi di sciopero), questo si verifica quando il modello estrae un dato errato nonostante quello corretto sia presente nel testo. Un esempio tipico è l'inversione delle date: il testo originale riporta «sciopero dalle 09:00 alle 17:00», ma il modello compila lo schema JSON inserendo come orario di fine le 13:00. Rientrano in questa categoria anche i gravi errori di classificazione (che nei nostri benchmark sono stati definiti *phantom strike*): il modello legge un avviso che parla di «revoca dello sciopero» o «adesione del personale allo sciopero precedente», ma lo classifica erroneamente come un nuovo sciopero in arrivo, contraddicendo il senso semantico del documento sorgente.
- **Allucinazione estrinseca (*extrinsic hallucination*)**: l'output contiene informazioni aggiuntive che non possono essere né verificate né contraddette partendo dalla sola fonte fornita in input. Come evidenziato da [JLF+23], una delle cause principali di questo errore è il *parametric knowledge bias*: il modello tende a dare priorità alla conoscenza pregressa memorizzata nei suoi pesi du-

rante l'addestramento, ignorando i limiti del contesto fornito. Nello scraping di avvisi sindacali, questo si verifica tipicamente quando la fonte è lacunosa. Se un testo riporta «sciopero venerdì 27 marzo» omettendo l'anno (perché rimosso da un preprocessing troppo aggressivo o assente nella pagina), il modello potrebbe inserire arbitrariamente 2024 o 2025 basandosi su un'ipotesi probabilistica – fenomeno emerso durante gli esperimenti e discusso in sezione 5.3.2. Allo stesso modo, potrebbe «indovinare» fasce di garanzia tipiche (es. «06:00-09:00») attingendo dalla sua conoscenza pregressa sul trasporto pubblico italiano, anche se l'avviso specifico non ne faceva menzione.

2.4.2 Perché le allucinazioni sono un problema critico nello scraping

Nello scraping tradizionale, basato su selettori CSS, XPath o espressioni regolari, l'errore ha una natura binaria e deterministica. Se la pagina web cambia struttura o il testo non rispetta il pattern atteso, lo scraper fallisce in modo esplicito: restituisce un valore nullo, una stringa vuota, o genera un'eccezione a runtime. Questo comportamento, noto nell'ingegneria del software come principio del «fail-fast», rappresenta paradossalmente una garanzia di sicurezza. Il sistema si interrompe prima di propagare l'errore, permettendo l'attivazione di meccanismi di allerta e l'intervento umano per l'aggiornamento del codice.

Con l'introduzione degli LLM, questo paradigma si capovolge radicalmente. Come evidenziato dalle definizioni di allucinazione intrinseca ed estrinseca, l'errore dell'intelligenza artificiale non si manifesta quasi mai come un'interruzione del processo. Un modello generativo è intrinsecamente progettato per produrre il completamento testuale statisticamente più probabile; di conseguenza, raramente «si arrende» di fronte a un input ambiguo, frammentato o privo delle informazioni necessarie. Al contrario, l'LLM tende a colmare i vuoti interpretativi forzando associazioni errate o attingendo alla propria conoscenza parametrica.

La criticità di questo comportamento esplode quando gli LLM vengono impiegati per l'estrazione di dati strutturati. I modelli moderni sono diventati eccellenti nel rispettare rigorosamente gli schemi di output richiesti, come le strutture JSON. Il risultato di un'allucinazione è quindi un oggetto dati formalmente ineccepibile: i tipi di dato sono rispettati, i campi obbligatori sono valorizzati e la validazione sintattica ha pieno successo. Per i sistemi informatici a valle (database, interfacce utente, motori di notifica), un dato allucinato risulta a tutti gli effetti indistinguibile da un'estrazione corretta.

Questo fenomeno trasforma il profilo di rischio dell'intera infrastruttura di scraping. Si passa da un problema di *parsing failure* – oneroso in termini di manutenzione ma

facilmente intercettabile – a un problema di *data integrity* silenzioso. Nel contesto di un sistema informativo rivolto al pubblico, le conseguenze sono dirette: mentre un fallimento deterministico porta a un'omissione temporanea del servizio (un ritardo nella pubblicazione di una notizia), un'allucinazione genera disinformazione attiva. La classificazione erronea di un report statistico come nuovo sciopero, o l'invenzione di una data non presente nel testo, minano alla base l'affidabilità e lo scopo stesso dell'applicazione.

L'adozione degli LLM, pertanto, non elimina la complessità ingegneristica dell'estrazione dati, ma ne sposta il baricentro. La sfida non risiede più nel mantenere aggiornati i selettori per estrarre le stringhe, ma nel progettare un'architettura in grado di arginare, rilevare e mitigare le allucinazioni semantiche prodotte dal modello.

2.4.3 Tecniche di mitigazione

La gestione delle allucinazioni richiede l'adozione di strategie mirate lungo l'intera pipeline di estrazione. Come evidenziato da Ji et al. [JLF+23], una delle cause primarie di questo fenomeno risiede nella presenza di rumore semantico e informazioni eterogenee nei dati forniti in input. In quest'ottica, le tecniche di preprocessing descritte nella sezione 2.3 non servono solo a ridurre i costi dovuti al numero di token in input: pulire il contesto dal rumore informativo aiuta anche a ridurre le allucinazioni estrinseche, limitando la possibilità che il modello venga fuorviato da dati contraddittori o irrilevanti.

Oltre all'intervento sull'input, la letteratura individua nelle tecniche di post-processing uno strumento essenziale per la correzione e la mitigazione delle allucinazioni [JLF+23]. Partendo da questo presupposto teorico, nel presente lavoro è stata progettata e sperimentata una specifica architettura di post-elaborazione e validazione (definita in seguito come approccio «lenient»), volta a separare l'estrazione probabilistica dell'IA dalla normalizzazione deterministica del dato. Le dinamiche di questa soluzione e la sua valutazione sperimentale saranno approfondite rispettivamente nella sezione 3.5 e nel capitolo 5.

2.5 Output strutturato e carico cognitivo

Oltre all'estrazione dell'informazione dal testo grezzo, l'ultimo miglio di una pipeline di web scraping semantico consiste nel costringere il modello a restituire i dati in un formato strutturato e facilmente processabile (tipicamente JSON), affinché possano essere facilmente integrati in un database.

Storicamente, ottenere un JSON sintatticamente valido da un LLM rappresentava una sfida notevole, spesso affrontata tramite tecniche di *prompt engineering* e

un lavoro di post-processing per correggere errori di formattazione. Più recentemente, i principali provider di IA (OpenAI, Google, ecc.) hanno introdotto una modalità di *Structured Output* nelle proprie API, che forzano il rispetto di uno schema JSON predefinito. Questo ha parzialmente risolto il problema della validità sintattica [BDW26].

Tuttavia, la garanzia sintattica non risolve un problema di natura semantica: il **carico cognitivo** imposto al modello. Nei domini aziendali reali, i dati estratti devono spesso aderire a standard rigorosi: date formattate in ISO 8601, stringhe mappate su enum o località tradotte in codici standard (come, nel nostro caso, i codici ISTAT per le province italiane).

In letteratura si sta affermando un dibattito metodologico su dove posizionare il confine tra l'inferenza probabilistica dell'IA e l'esecuzione deterministica del software classico. In termini di architettura dei dati, questo dibattito riflette la scomposizione di una tipica pipeline ETL. Richiedere all'LLM di estrarre il dato e contemporaneamente normalizzarlo secondo rigide regole di business (approccio *strict*) significa delegargli sia l'«extract» che il «transform», aumentando la probabilità di fallimento logico, specialmente nei modelli di dimensioni più contenute (come SLM) che dispongono di una minore capacità di ragionamento. Di contro, un approccio più tollerante (*lenient*) delega all'LLM esclusivamente l'estrazione semantica del «testo libero» (es. estrarre «Roma» o «Domani pomeriggio»), demandando la fase di normalizzazione (il «transform») a classici script deterministici eseguiti a valle.

Comprendere quale di questi due approcci bilanci meglio accuratezza e costi computazionali è uno dei quesiti di ricerca affrontati sperimentalmente nel capitolo 5.

2.6 Il limite della latenza

Mentre l'integrazione degli LLM risolve la fragilità strutturale dei parser tradizionali, questa introduce anche un limite prestazionale: la latenza. Come accennato nella sezione 2.1, i wrapper tradizionali basati su selettori DOM (XPath o CSS) risolvono l'estrazione in modo deterministico in una frazione di secondo.

Al contrario, l'estrazione semantica tramite modelli generativi impone tempi di esecuzione ordini di grandezza superiori. Un LLM deve prima elaborare l'intero contesto in input (il questo caso, il documento web pre-processato) e successivamente generare l'output (il file JSON) in modo *autoregressivo*, ossia producendo una parola/token alla volta in sequenza, basando ogni nuova predizione su quelle precedenti. Questo processo dilata la latenza, portandola nell'ordine dei secondi o, in alcuni casi, delle decine di secondi. Se questo ritardo è tollerabile in pipeline di scraping

asincrone (es. l'aggiornamento notturno di un database), risulta invece proibitivo in scenari *real-time*.

Storicamente, il problema risiede in un disallineamento tra l'hardware a disposizione e la natura del task. Le infrastrutture di inferenza IA standard, basate su GPU, sono progettate per massimizzare il *throughput* complessivo aggregando e processando molte richieste simultaneamente (una tecnica denominata «batching»). Malgrado ciò, il web scraping su richiesta è intrinsecamente un'operazione con *batch-size* pari a 1: il sistema deve analizzare un singolo documento e restituire il risultato il più velocemente possibile, uno scenario in cui le GPU tradizionali risultano inefficienti.

Per superare questo collo di bottiglia, l'industria sta sviluppando architetture hardware concepite specificamente per l'inferenza a bassissima latenza. Un contributo importante in questa direzione è l'architettura TSP (*Tensor Streaming Processor*, spesso commercializzata come LPU, *Language Processing Unit*), introdotta da Groq [ARS+20]. Senza entrare nei dettagli microarchitetturali, la peculiarità di questi chip risiede nell'eliminazione delle complesse gerarchie di memoria e di scheduling dinamico tipiche delle GPU, in favore di un'esecuzione dei calcoli tensoriali strettamente deterministica.

L'impatto applicativo di questa innovazione per il dominio dell'estrazione dati è notevole: questi processori sono ottimizzati esattamente per raggiungere velocità estreme su singole richieste (scenari a batch-size di 1). Come verrà evidenziato empiricamente nel capitolo 5, l'esecuzione di modelli linguistici di dimensioni modeste su hardware TSP (come Llama 4 Scout da 17 miliardi di parametri) consente di ridurre la latenza a pochi secondi.

2.7 Valutazione dell'estrazione

Nello scraping tradizionale, il successo di un'estrazione era spesso valutato in modo binario: se il selettore XPath o la *regex* catturavano o meno la stringa esatta. Con l'estrazione semantica tramite LLM, dove l'output è strutturato (tipicamente JSON con campi multipli) e soggetto a variazioni di formato e allucinazioni, la letteratura ha adottato metriche più articolate, mutuate dal dominio dell'NLP.

Le metriche di riferimento consolidate sono la *precision* (precisione), la *recall* (richiamo) e la loro media armonica, l'*F1-score*. La prima penalizza allucinazioni e dati inventati (falsi positivi), mentre la seconda penalizza le omissioni di informazioni presenti nel documento (falsi negativi). Come evidenziato da Kim et al. [KKJ25], l'impiego di queste metriche nasce dalla necessità di assegnare un «credito parziale» all'estrazione, gestendo così i casi di fallimento parziale.

In task complessi, un modello potrebbe infatti identificare correttamente gran parte delle informazioni di un record, commettendo errori solo su una porzione (ad esempio, omettendo una data o allucinando un singolo campo). In questi scenari, metriche basate sulla corrispondenza esatta dell'intero blocco (*exact match*) risulterebbero eccessivamente punitive e poco rappresentative della reale capacità di comprensione del modello [BMR+20]. L'uso dell'F1-score permette invece di valutare la correttezza dei singoli attributi individuati, superando la logica binaria dello scraping tradizionale e offrendo una misura più rappresentativa dell'accuratezza del dato estratto. L'applicazione di queste metriche nel presente lavoro è descritta nella sezione 5.1.

Capitolo 3

Disegno sperimentale e metodologia di valutazione

Per rispondere ai quesiti di ricerca sollevati nel capitolo precedente, è stato progettato un ambiente di test controllato: un framework di benchmark. Il dominio scelto per la valutazione sperimentale è quello degli avvisi di sciopero nel settore del trasporto pubblico italiano. La natura eterogenea di questi documenti (strutturati, semi-strutturati e testo libero) permette infatti di stressare le capacità di generalizzazione degli LLM e di quantificare concretamente l'affidabilità e i costi associati a diverse strategie di estrazione.

3.1 Struttura del dataset e schema di ground truth

La validità di un benchmark dipende intrinsecamente dalla qualità dei dati su cui viene eseguito. Per questo motivo, il dataset utilizzato per la sperimentazione è composto esclusivamente da avvisi storici realmente pubblicati dalle aziende di trasporto. Si è scelto deliberatamente di non fare ricorso a dati generati sinteticamente per garantire la massima validità dell'esperimento e per evitare l'introduzione di bias sistematici che avrebbero potuto favorire artificiosamente gli approcci basati su LLM.

Il dataset è accompagnato da un file di ground truth in formato JSON, costruito manualmente con un processo di validazione assistita da IA. Per ogni documento, è stata generata una prima bozza di estrazione utilizzando un modello linguistico (Gemini 3.1 Pro), che è stata poi verificata e corretta a mano. Questo approccio ha permesso di accelerare significativamente la fase di annotazione, mantenendo al contempo l'accuratezza dei dati di riferimento.

Lo schema di ciascun record prevede prima una classificazione: se il documento riguarda l'annuncio di un nuovo sciopero o meno (es. una revoca o un report di adesione). Per i documenti classificati come annunci di sciopero, vengono annotati i campi comparabili, ossia data di inizio e fine, ambito territoriale (nazionale o regioni specifiche) e fasce orarie garantite. L'ambito territoriale, se non nazionale, include anche un campo sulle regioni coinvolte.

Questi campi presentano livelli di complessità diversi: le date richiedono l'interpretazione di riferimenti temporali relativi (es. «domani»), l'ambito territoriale può essere implicito (ad esempio, gli avvisi di ATAC raramente specificano che lo sciopero è limitato a Roma), la lista di regioni coinvolte deve essere codificata secondo un enum predefinito (i codici ISTAT) e le fasce orarie compaiono spesso in formato tabellare e con espressioni eterogenee (es. «dalle 8 alle 17», «tutto il giorno»).

Questa traduzione da linguaggio naturale a formato programmatico rappresenta una sfida notevole per i modelli. Come si vedrà nella sezione 3.5, tale complessità può essere gestita rilassando i vincoli dello schema di risposta dell'LLM e delegando la normalizzazione finale a codice deterministico (approccio *lenient*).

3.2 Disegno sperimentale e variabili di valutazione

I benchmark sono stati eseguiti tramite **SWEET**, progettato per testare in modo sistematico le diverse configurazioni del sistema d'estrazione. Il disegno sperimentale segue un approccio combinatorio: per ogni esecuzione viene generata una matrice che incrocia i modelli LLM con le strategie di preprocessing, le tecniche di prompting e gli schemi di validazione.

Per facilitare l'analisi dei risultati, i test sono stati organizzati in suite specifiche corrispondenti alle diverse fonti dei documenti (come Trenord, Trenitalia, EAV, ATAC). Questa suddivisione permette di identificare le performance di ciascuna fonte e di valutare l'efficacia delle strategie adottate in contesti reali e diversificati.

L'esecuzione dei test segue un protocollo che astrae dalle complessità architetturali dell'infrastruttura (come la gestione della cache, la concorrenza delle API e i meccanismi di tolleranza agli errori, discussi in dettaglio nel capitolo 4), per concentrarsi sulla misurazione oggettiva dei risultati.

La valutazione di ogni esperimento si basa sul confronto deterministico tra l'output generato dall'LLM e il ground truth. In questa fase vengono misurate due categorie di variabili:

- Le metriche relative all'efficacia e all'accuratezza dell'estrazione, ovvero precision, recall e F1-score. La scelta di queste metriche verrà approfondita nella sezione 5.1.
- Le metriche di sostenibilità operativa, come il costo stimato in base ai token consumati e la latenza di esecuzione.

L'aggregazione di questi dati permette di quantificare concretamente il compromesso tra l'affidabilità del dato estratto e i costi associati alle diverse architetture proposte.

3.3 Strategie di preprocessing valutate

Le pagine HTML, pensate per la fruizione umana, sono spesso estremamente verbose: contengono script, stili, elementi di navigazione e attributi che, se inviati direttamente al modello, si traducono in un consumo elevato di token (spesso decine di migliaia di token per pagina) e introducono rumore che può confondere l'estrazione. Una riduzione intelligente del contenuto, che preservi le informazioni semanticamente rilevanti, può abbattere i costi di uno o due ordini di grandezza e migliorare l'accuratezza, ma deve essere operata con cautela: un preprocessing troppo aggressivo rischia di eliminare elementi utili alla comprensione, come i meta-tag o la struttura delle tabelle. In questo studio sono state confrontate sei strategie, che variano da approcci puramente deterministici a soluzioni basate su modelli linguistici di piccole dimensioni (Small Language Model).

HTML grezzo (`raw-html`) La strategia più semplice consiste nell'inviare all'LLM il contenuto HTML originale della pagina, senza alcuna modifica. Questa modalità funge da punto di riferimento per valutare il contributo delle successive ottimizzazioni. Nella pratica, però, l'HTML grezzo è spesso troppo voluminoso e ricco di elementi non testuali (script, stili, commenti) che aumentano il costo senza apportare benefici. Per questo motivo, nei benchmark principali questa strategia è stata utilizzata solo come riferimento e non inclusa nelle valutazioni comparative di costo.

Pulizia di base dell'HTML (`basic-cleanup`) Questa strategia applica una serie di trasformazioni deterministiche per ridurre il rumore mantenendo la struttura semantica della pagina. Vengono rimossi tutti i tag che non contengono testo utile alla comprensione, come `<script>`, `<style>`, `<noscript>`, `<iframe>`, `<canvas>` e simili. Inoltre, la maggior parte degli attributi dei tag (classi, ID, stili) viene eliminata, fatta eccezione per quelli che possono contenere informazioni rilevanti, come gli `href` dei link o gli attributi che strutturano tabelle. Il risultato è un HTML molto più snello ma ancora fedele alla gerarchia originale, che costituisce una solida base per le elaborazioni successive.

Conversione in Markdown (`html-to-markdown`) Partendo dall'HTML pulito, questa strategia lo converte in formato Markdown. La scelta è motivata dalla maggiore compattezza del Markdown rispetto all'HTML e dalla sua familiarità per molti modelli, addestrati su enormi quantità di testo in questo formato. La conversione preserva gli elementi strutturali essenziali (titoli, liste, tabelle, enfasi) attraverso una sintassi leggera, con un numero ridotto di token ma ancora semanticamente ricca.

Distillazione del DOM (`dom-distillation`) A differenza delle strategie precedenti, completamente automatiche, la distillazione del DOM introduce una componente manuale. Per ogni fonte, vengono identificati a priori uno o più selettori CSS che individuano i contenitori principali delle informazioni (ad esempio, l'elemento che racchiude l'intero avviso di sciopero). Lo scraper estrae solo questi contenitori, scartando il resto (come header, footer, menu laterali e altri elementi di contorno). Non si tenta di estrarre singoli campi (come la data o il titolo), ma si isolano blocchi più ampi che contengono tutto il testo rilevante. Questo approccio ibrido richiede un investimento umano iniziale, ma è particolarmente efficace quando le fonti hanno una struttura stabile nel tempo, come spesso accade nei siti di aziende di trasporto. Il guadagno in termini di riduzione dei token è notevole, e l'assenza di rumore contestuale riduce il rischio di allucinazioni.

Jina Reader (`jina-reader`) Jina Reader è un servizio basato su un modello specializzato (ReaderLM-v2, da 1,5 miliardi di parametri) progettato per convertire HTML in Markdown o JSON «pulito». A differenza delle strategie puramente euristiche, questo approccio sfrutta un Small Language Model (SLM) addestrato specificamente per l'estrazione di contenuto da pagine web, gestendo documenti lunghi fino a 512.000 token e oltre 20 lingue. L'uso di un modello dedicato promette di ottenere una pulizia più intelligente, preservando le informazioni semantiche anche in presenza di markup complessi. Jina Reader è stato valutato come rappresentante degli approcci commerciali basati su SLM.

MinerU HTML (`mineru-html`) MinerU (o Dripper) è un modello open-source, anch'esso basato su un SLM (derivato da Qwen3), specializzato nell'estrazione del contenuto principale da pagine HTML e nella sua restituzione in forma minimizzata. Rispetto a Jina Reader, MinerU è progettato per una compressione più aggressiva, mirando a eliminare quasi tutto il boilerplate e lasciare solo il testo essenziale. I benchmark mostrano che può ridurre la dimensione dell'input di oltre il 97%, ma con il rischio di perdere informazioni contestuali importanti, quali i meta-tag o l'anno in date espresse in modo implicito, come dimostrato in sezione 5.3.2. Questa strategia rappresenta l'estremo dello spettro efficienza/accuratezza.

Per le strategie che richiedono modelli dedicati (`jina-reader` e `mineru-html`), il preprocessing è stato eseguito una tantum e i risultati salvati su disco, in modo da poter essere riutilizzati per più run di benchmark senza ripetere l'elaborazione, che sarebbe risultata costosa in termini di tempo o di risorse computazionali.

3.4 Modelli LLM testati

La selezione dei modelli ha coperto diverse famiglie e fasce di costo, con l'obiettivo di confrontare approcci eterogenei sia per architettura che per politica di prezzo:

- GPT-5 nano di OpenAI, che rappresenta la fascia più economica e accessibile tra i modelli GPT, con prestazioni inferiori ma costi molto contenuti.
- Gemini 3.1 Flash Lite, rilasciato a marzo 2026, un modello di Google di fascia economica, progettato per attività come «l'estrazione semplice di dati» a bassa latenza.
- Llama-4 Scout 17B, modello open-weight di grandi dimensioni, parte della quarta generazione di modelli aperti di Meta.
- DeepSeek Chat (DeepSeek-V3.2), sviluppato da DeepSeek, un altro modello open-weight di grandi dimensioni.

Per ciascun modello sono state registrate le tariffe aggiornate a marzo 2026 (si veda la tabella 3.1), utilizzate per il calcolo dei costi presentato nel capitolo 5. L'integrazione con i diversi provider è stata realizzata in modo da uniformare il più possibile le modalità di interazione, minimizzando le differenze dovute all'infrastruttura e concentrando l'analisi sulle performance intrinseche dei modelli.

Modello	**Provider**	**Tipologia**	**Costo input** *(1M token)*	**Costo output** *(1M token)*
GPT-5 nano	OpenAI	Proprietario	$ 0,050	$ 0,400
Gemini 3.1 Flash Lite	Google	Proprietario	$ 0,250	$ 1,500
Llama-4 Scout 17B	Meta (via Groq)	Open-weight	$ 0,110	$ 0,340
DeepSeek-V3.2 (Chat)	DeepSeek	Open-weight	$ 0,028	$ 0,280

Tabella 3.1: Modelli LLM selezionati per il benchmark e relative tariffe API (aggiornate a marzo 2026). I costi sono espressi in dollari statunitensi (USD).

3.5 Schema di validazione: *strict* vs *lenient*

Un importante aspetto metodologico valutato in questa ricerca riguarda il confine tra l'estrazione probabilistica (delegata all'LLM) e la normalizzazione deterministica (delegata al codice tradizionale). Sono state messe a confronto due filosofie opposte:

- **Validazione «strict» (rigida)**: in questo approccio, si richiede al modello di generare un output che sia già perfettamente conforme allo schema finale del database. Ciò implica che l'LLM debba non solo comprendere il testo, ma anche applicare regole di dominio rigide (ad esempio, mappare «Emilia-Romagna» nel codice ISTAT di regione «08», o formattare una data esattamente in `yyyy-MM-dd HH:mm:ss`). Qualsiasi deviazione da queste regole comporta il fallimento dell'estrazione per invalidazione dello schema.
- **Validazione «lenient» (tollerante)**: in questo approccio, si rilassano i vincoli imposti al modello. All'LLM viene permesso di estrarre le informazioni in formato «grezzo» (ad esempio, restituendo il nome esteso della città o una data in un formato processabile da codice). Il compito di normalizzare questi dati imperfetti e ricondurli a standard rigidi viene delegato a valle, a un post-processore deterministico basato su euristiche tradizionali (come dizionari di lookup e parsing di date).

L'obiettivo di questo confronto è verificare se alleggerire il «carico cognitivo» dell'LLM riduca il tasso di allucinazioni e migliori la *recall*, specialmente nei modelli di dimensioni più contenute.

3.6 Confronto tra istruzioni generiche ed esplicite

L'interazione con un LLM non è determinata solo dal modello utilizzato, ma in misura critica dalle istruzioni fornite nel *prompt*. Un'ipotesi centrale di questo lavoro è che la capacità del modello di classificare correttamente un contenuto rilevante (in questo caso, uno sciopero futuro) da contenuti simili ma non pertinenti (revoche, report di adesione passati) dipenda dalla granularità delle regole di dominio fornite in linguaggio naturale.

Per verificare questa ipotesi, sono state definite e messe a confronto due strategie di prompting distinte, i cui testi integrali sono riportati nell'appendice A:

1. **Istruzione generica**: in questa configurazione, il prompt si limita a definire il ruolo dell'agente («Sei un algoritmo di estrazione dati») e a richiedere la compilazione dello schema JSON. Le regole fornite riguardano quasi esclusivamente il *formato* dei dati (es. «le date devono essere in formato ISO», «usa

i codici ISTAT»). Si assume implicitamente che il modello possieda una comprensione sufficiente del concetto di *sciopero* per discriminare autonomamente i casi positivi da quelli negativi.

2. **Istruzione esplicita**: in questa configurazione, la logica di business che tradizionalmente verrebbe scritta in codice (es. `if (text.contains("revoca")) return false`) viene trasposta in regole semantiche all'interno del prompt. L'istruzione viene arricchita con:

   - **Regole decisionali esplicite**: viene chiesto al modello di eseguire un primo passaggio logico per decidere la natura del documento (es. «Decidi se questo documento annuncia effettivamente un nuovo sciopero»). Questo prende anche il nome di *chain of thought prompting*, dove il modello è guidato a ragionare passo dopo passo prima di fornire la risposta finale.
   - **Vincoli negativi**: vengono elencati casi specifici di esclusione che potrebbero generare ambiguità (es. «Se è una revoca, imposta isStrike a false», «Se riporta dati di adesione passati, imposta isStrike a false»).
   - **Few-shot prompting**: vengono forniti esempi concreti di output JSON attesi sia per il caso positivo (sciopero rilevato) che per il caso negativo (non sciopero), per guidare il modello verso la struttura desiderata tramite l'apprendimento contestuale.

L'obiettivo di questo confronto è quantificare se e quanto l'investimento in un prompt più verboso (e quindi più costoso in termini di token di input) sia giustificato da un aumento della precisione nella classificazione, specialmente in domini caratterizzati da un linguaggio burocratico ambiguo.

3.7 Disegno sperimentale per il test di resilienza

Uno dei quesiti centrali della ricerca mira a quantificare il vantaggio competitivo degli approcci semantici rispetto a quelli deterministici a fronte della naturale evoluzione del web. Per rispondere a questa domanda, si è reso necessario definire un protocollo di valutazione che isoli la variabile della **struttura sintattica** dalla variabile del **contenuto informativo**.

L'obiettivo della metodologia qui proposta è misurare il grado di dipendenza di ogni strategia di estrazione dalla forma del documento sorgente. Il disegno sperimentale prevede una procedura di confronto basata su due versioni dello stesso dataset:

- Il dataset originale, contenente i documenti così come acquisiti dalle fonti, con la loro struttura HTML e le classi CSS originali.

- Il dataset modificato, una versione alterata degli stessi documenti: la struttura DOM è stata intenzionalmente modificata per eliminare ogni riferimento sintattico, lasciando però del tutto invariato il testo visibile a un utente umano.

L'alterazione controllata del dataset mira a riprodurre due fenomeni critici del web scraping reale: le modifiche di layout, che si manifestano attraverso il cambiamento di classi, ID e tag; e le tecniche di offuscamento anti-scraping, che introducono rumore negli attributi e frammentano la struttura dei nodi.

Sotto il profilo metodologico, l'esperimento consiste nell'eseguire l'intera matrice di test su entrambi i dataset e misurare la variazione delle performance. Mentre uno scraper tradizionale si basa sull'ipotesi che la posizione e l'identità sintattica di un elemento siano stabili, l'ipotesi di ricerca prevede che un approccio basato su LLM sia in grado di estrarre le informazioni basandosi unicamente sulla semantica testuale.

I risultati di questo confronto forniranno una misura empirica della resilienza di ciascun approccio, evidenziando in che misura la dipendenza dalla struttura sintattica rappresenti un punto di vulnerabilità per i sistemi di web scraping tradizionali.

Capitolo 4

Implementazione di SWEET

Il capitolo descrive l'implementazione di **SWEET** (*Semantic Web Extraction & Evaluation Tool*), fornendo gli elementi necessari per comprenderne il funzionamento e, eventualmente, replicare gli esperimenti.

Nella prima sezione si illustra come è stato assemblato il dataset sperimentale, a partire dall'archiviazione di fonti eterogenee e dalla costruzione del ground truth. Successivamente si presenta l'architettura complessiva del modulo `BenchmarksModule` e il suo flusso di esecuzione, per poi scendere nel dettaglio delle componenti chiave: il pattern *adapter* per l'astrazione delle API di IA, il pattern *strategy* per il preprocessing, il motore di esecuzione con il sistema di caching, l'algoritmo di scoring e il generatore di «DOM chaos» per i test di resilienza.

4.1 Costruzione e archiviazione del dataset

Affinché il motore di benchmark descritto nel capitolo 3 potesse operare in modo deterministico e riproducibile, è stato necessario predisporre un'infrastruttura di archiviazione locale dei dati grezzi, isolando gli esperimenti dalla volatilità del web.

L'acquisizione materiale dei documenti è derivata da due filoni paralleli. Il primo è costituito da un sistema di monitoraggio sviluppato contestualmente a questo lavoro (denominato «C'è Sciopero?»), che ha operato in produzione per diversi mesi registrando e salvando automaticamente le pagine HTML e i PDF dei nuovi avvisi man mano che venivano pubblicati dalle aziende di trasporto.

Il secondo filone è consistito in un'attività di estrazione storica. Per i siti web statici provvisti di archivi accessibili (come Trenord), sono stati realizzati degli script di scraping ad-hoc che hanno automatizzato il download massivo delle pagine passa-

te, permettendo di risalire storicamente fino agli avvisi del 2022. Per le fonti che pubblicano esclusivamente in formato PDF (come Trenitalia TPER), è stata impiegata l'automazione browser per superare i blocchi basati su JavaScript e scaricare programmaticamente i file binari.

Questa operazione congiunta ha permesso di costituire un archivio locale di 234 documenti unici. Sebbene in ambito di machine learning si operi solitamente con dataset di ordini di grandezza superiori, in questo specifico dominio la dimensione dell'archivio è vincolata dalla natura stessa dell'evento: gli scioperi sono episodi relativamente rari e le aziende di trasporto tendono a rimuovere tempestivamente gli avvisi dai propri portali una volta conclusa l'agitazione, rendendo complessa la costituzione di una base dati estesa. Tuttavia, l'elevata eterogeneità dei formati raccolti si è rivelata ampiamente sufficiente per stressare e valutare le pipeline di estrazione.

A livello di file system, il dataset è stato organizzato in una struttura gerarchica di directory suddivise per azienda di trasporto (ad esempio, `data/Trenord/` e `data/ATAC/`), ciascuna contenente esclusivamente i documenti grezzi originali. Nella directory principale `data/` è presente il file `ground-truth.json`; tale file mappa l'identificativo di ogni documento locale alla sua rappresentazione strutturata attesa, fungendo da sorgente di verità per il modulo di *scoring* del benchmark.

4.2 Architettura di SWEET e design pattern

Come illustrato nella figura 4.1, il sistema è progettato attorno a un flusso di lavoro modulare che separa nettamente la preparazione dei dati, l'orchestrazione dei test e l'esecuzione materiale delle chiamate ai modelli. Il processo prende in input il dataset di documenti originali e le loro versioni alterate, genera una matrice di test combinando fonti, modelli e strategie di preprocessing, e confronta deterministicamente gli output con il ground truth per calcolare lo scoring finale.

L'infrastruttura si appoggia su un ecosistema di tecnologie e librerie consolidate (riassunte nella tabella 4.1), selezionate per garantire l'estensibilità del framework e la precisione nella manipolazione dei documenti web e dei dati strutturati.

4.2.1 Panoramica del modulo

L'infrastruttura di SWEET è realizzata come un'applicazione standalone in TypeScript[1], volutamente progettata come un'entità separata e leggera rispetto al sistema

[1] Il codice sorgente completo, il dataset e gli strumenti di analisi sono disponibili pubblicamente su GitHub all'indirizzo: `https://github.com/alessandroamella/llm-scraping-benchmarks`

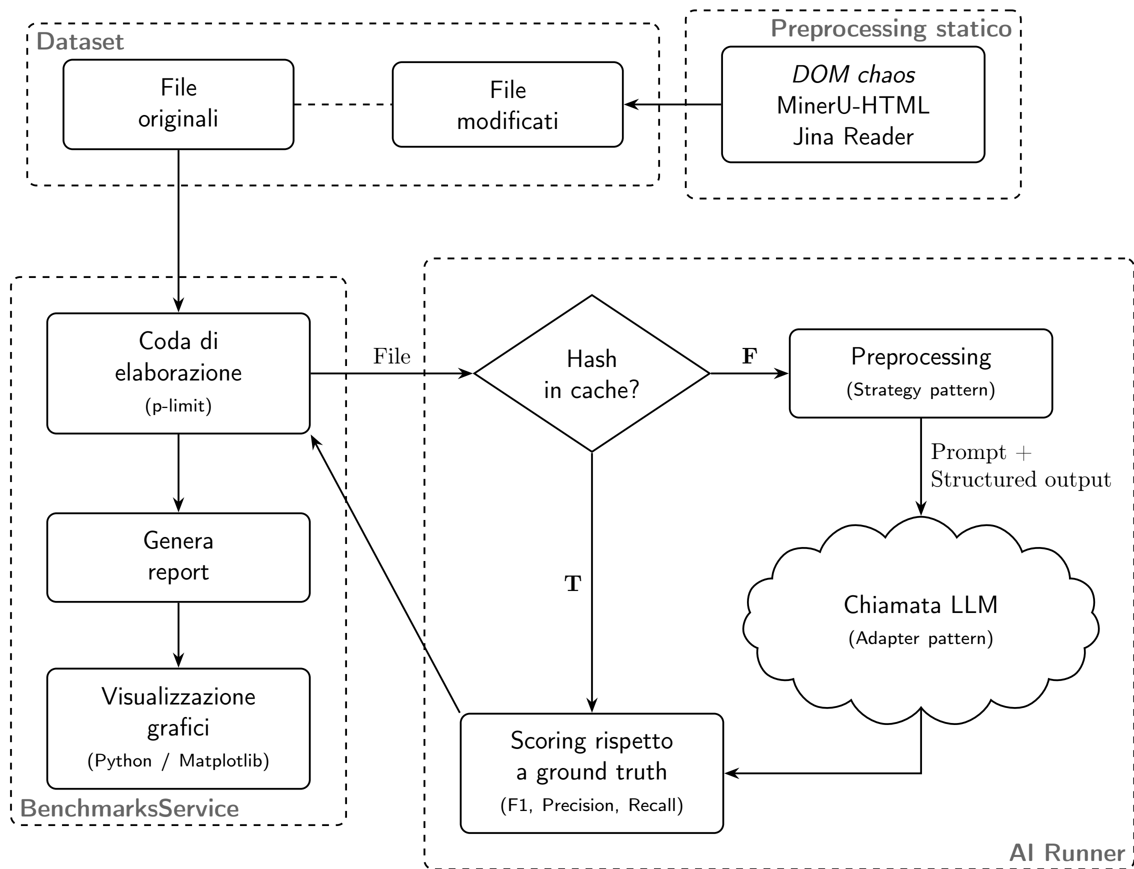

Figura 4.1: Diagramma del flusso architetturale di SWEET.

Ambito	Libreria / Tool	Ruolo nel Framework
Infrastruttura	NestJS (su Bun.sh)	Framework per l'architettura a moduli e la dependency injection.
	p-limit	Gestione della coda di esecuzione e controllo della concorrenza (RPM/TPM).
Parsing documenti	Cheerio	Manipolazione del DOM, pulizia dell'HTML, *distillation* e generatore di caos.
	Turndown	Conversione euristica da HTML a Markdown (compatibile con GFM).
	pdf-parse	Estrazione del testo grezzo dai documenti PDF (es. Trenitalia TPER).
Integrazione AI	OpenAI / Google SDK	Interfacciamento con i modelli Gemini e compatibili con la SDK di OpenAI.
	Tiktoken	Conteggio accurato dei token per i modelli di OpenAI.
	jsonrepair	Recupero e riparazione sintattica di output JSON malformati.
Validazione	Zod	Definizione degli schemi (*strict/lenient*) e validazione dei dati estratti.
	Lodash	Operazioni insiemistiche per il calcolo di TP, FP e FN sui vettori.
Analisi dati	Pandas	Elaborazione statistica dei report JSON prodotti dai benchmark.
	Matplotlib / Seaborn	Generazione dei grafici di accuratezza, costo e resilienza.

Tabella 4.1: Stack tecnologico e librerie principali utilizzate nel framework di benchmark.

di produzione da cui provengono i dati, in cui la logica del motore di benchmark risiede nel modulo `BenchmarksModule`.

Questa scelta deriva dagli obiettivi della sperimentazione: eseguire decine di migliaia di chiamate a API esterne, manipolare i dati in modi potenzialmente distruttivi (si pensi al generatore di caos) e iterare rapidamente sulle metriche di valutazione. Un'applicazione dedicata garantisce flessibilità, consente di utilizzare strumenti ottimizzati e rende il framework riproducibile: chiunque può clonare il repository, configurare le proprie chiavi API in un file `.env` ed eseguire gli stessi identici test senza dover mettere in piedi alcuna infrastruttura aggiuntiva.

Come dettagliato nella sezione 4.3, l'orchestrazione si basa su una matrice combinatoria che incrocia modelli e strategie. Per ogni singola esecuzione, il sistema applica una pipeline a quattro fasi:

1. preprocessing del dato grezzo;
2. invocazione del modello LLM tramite l'adapter dedicato;
3. normalizzazione e validazione della risposta;
4. calcolo delle metriche di accuratezza (*scoring*).

I risultati, arricchiti da metadati tecnici come il consumo di token e la latenza, vengono infine aggregati in un report JSON che costituisce la base per le analisi statistiche del capitolo 5.

L'esecuzione parallela di migliaia di test è gestita tramite un meccanismo di code a concorrenza limitata, necessario per rispettare i rate limit imposti dai provider di LLM e per evitare di incorrere in costi imprevisti. I risultati di ogni singolo test, completi di metadati (modello, strategia, accuratezza, costo stimato, latenza, timestamp), vengono accumulati in un report JSON finale.

L'architettura del modulo favorisce l'estensibilità: l'aggiunta di un nuovo modello richiede solo la scrittura di un nuovo adapter (come descritto nella sezione 4.2.2), mentre l'introduzione di una nuova strategia di preprocessing si traduce nell'implementazione di una nuova logica all'interno della funzione orchestratrice `applyStrategy`. Questo disaccoppiamento ha permesso di testare diverse configurazioni diverse senza dover modificare il codice del motore di esecuzione principale.

4.2.2 Astrazione delle API con il pattern adapter

Una delle sfide tecniche nello sviluppo del modulo benchmark è stata la gestione della frammentazione tra i provider di modelli linguistici. Ciascun fornitore (OpenAI, Google, Groq, DeepSeek) espone una propria API con SDK dedicati, formati di

risposta differenti e modalità eterogenee per il conteggio dei token e la stima dei costi. Integrare direttamente ogni SDK nel motore di benchmark avrebbe portato a una significativa duplicazione di codice, oltre a rendere l'aggiunta di un nuovo modello un'operazione laboriosa e soggetta a errori.

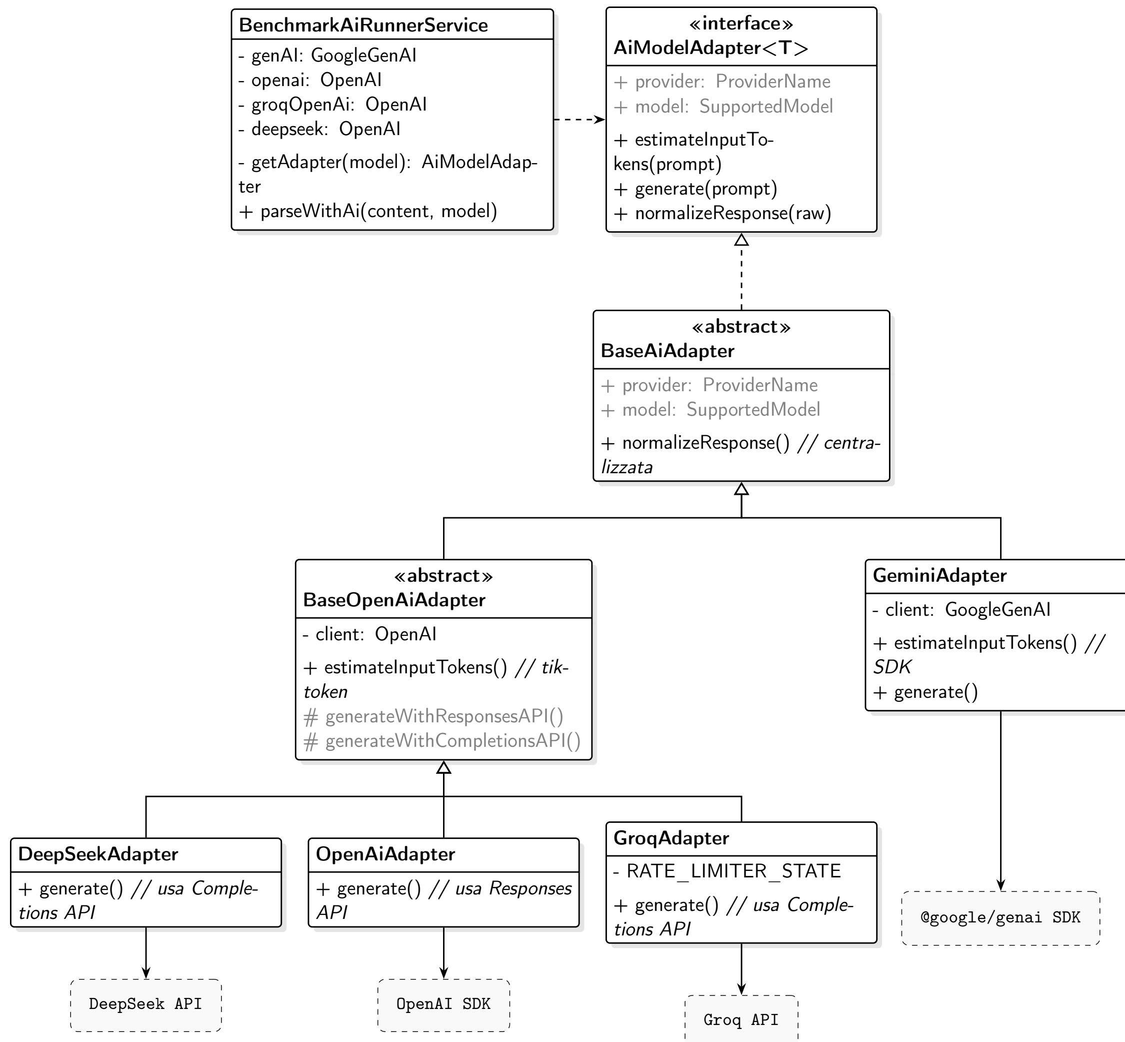

Figura 4.2: Diagramma delle classi (UML) del pattern adapter utilizzato per astrarre le chiamate ai provider di modelli linguistici. La struttura evidenzia la centralizzazione delle responsabilità (normalizzazione semantica e logiche API condivise) nelle classi astratte, minimizzando la duplicazione di codice nei client concreti.

Per risolvere questo problema, si è adottato il design pattern **adapter**. L'idea principale è definire un'interfaccia comune che tutte le implementazioni devono rispettare, nascondendo le specificità del provider all'interno delle rispettive classi. In questo modo, il motore di esecuzione (`BenchmarkAiRunnerService`) interagisce con i mo-

delli esclusivamente tramite un contratto prestabilito, rimanendo completamente agnostico rispetto al fornitore sottostante.

Struttura dell'interfaccia e gerarchia delle classi

L'interfaccia `AiModelAdapter<T>` definisce le tre operazioni necessarie all'orchestrazione del benchmark: `estimateInputTokens` per il calcolo preventivo dei costi e il monitoraggio dei limiti di contesto; `generate` per l'invocazione effettiva dell'API, incapsulando la gestione della rete (timeout, retry e parsing strutturale del JSON grezzo); e infine `normalizeResponse` per la riconciliazione semantica del dato.

Quest'ultimo passaggio è fondamentale ai fini del benchmark: non si occupa di aggiustare la sintassi JSON (già sanata a monte), ma di standardizzare i valori estratti per permettere un confronto coerente con la ground truth. Ciò include la traduzione di stringhe descrittive in codici ISTAT (es. «Emilia-Romagna» in «08»), la normalizzazione dei formati data e la mappatura delle variazioni linguistiche nei campi enumerati.

Come visibile in figura 4.2, l'architettura non si limita a un'interfaccia piatta, ma sfrutta l'ereditarietà per applicare rigorosamente il principio *DRY* (Don't Repeat Yourself). Piuttosto che duplicare la complessa logica di standardizzazione in ogni adapter, questa è stata centralizzata all'interno della classe astratta `BaseAiAdapter`. Da essa derivano tutte le implementazioni, garantendo che ogni modello venga valutato in modo equo attraverso il medesimo motore di regole semantiche.

Disaccoppiamento tramite gerarchia delle classi

Osservando la gerarchia del diagramma, la presenza di un ulteriore livello di astrazione (`BaseOpenAiAdapter`) risponde all'esigenza di gestire l'ecosistema frammentato ma standardizzato attorno al formato OpenAI. Molti provider (come DeepSeek e Groq) offrono API OpenAI-compatibili, rendendo ridondante riscrivere da zero le logiche di chiamata.

Questa classe intermedia condivide tra i modelli compatibili sia la stima dei token sia due strategie di generazione: la *Responses API* per output strutturati nativi e la *Completions API* tradizionale.

Le classi finali (i Concrete Adapters) si concentrano quindi esclusivamente sui vincoli specifici del singolo provider:

- **OpenAiAdapter** e **DeepSeekAdapter** demandano la chiamata ai metodi standardizzati della classe padre, selezionando rispettivamente la Responses API o la Completions API in base al grado di supporto degli Structured Outputs del provider.

- **GroqAdapter** si appoggia anch'esso all'infrastruttura OpenAI, ma vi inietta un rate limiter interno per far fronte alle stringenti policy del proprio fornitore.
- **GeminiAdapter**, appartenendo a un ecosistema completamente diverso, estende direttamente la base comune `BaseAiAdapter`, mascherando al suo interno le logiche del proprio SDK proprietario (`@google/genai`).

L'adozione di questa architettura basata sul pattern adapter ha profondamente migliorato la robustezza del modulo benchmark. Il motore centrale risulta del tutto disaccoppiato dalle evoluzioni (e dalle frequenti breaking changes) delle API esterne. Nuovi modelli possono essere introdotti agilmente derivando la classe astratta più pertinente, mentre la netta separazione delle responsabilità semplifica l'integrazione di sistemi di caching (sezione 4.3.3) e il calcolo unificato dei costi per l'intero sistema.

4.2.3 Pattern strategy per il preprocessing e il parsing

Il motore di benchmark si trova a dover orchestrare l'estrazione di dati da fonti eterogenee ricorrendo a metodologie radicalmente diverse: dall'estrazione deterministica tramite selettori CSS, all'uso di LLM con varie tecniche di pulizia dell'HTML, fino all'elaborazione di documenti PDF. Vincolare il motore centrale (`BenchmarksService`) alle specificità di ciascun estrattore avrebbe reso il sistema rigido e difficile da estendere.

Per disaccoppiare la logica di orchestrazione da quella di estrazione, si è adottato il design pattern **strategy**. Come illustrato in figura 4.3, il *context* interagisce con i parser esclusivamente tramite un contratto comune: l'interfaccia `IStrikeParser`. Questa espone un unico metodo asincrono (`parse`) che accetta in ingresso il contenuto grezzo della fonte (HTML o testo) e restituisce un oggetto standardizzato (`BenchmarkStrike`), incapsulando al proprio interno l'intera complessità dell'estrazione.

Le implementazioni concrete di questa interfaccia si dividono in tre categorie architetturali, ciascuna progettata per rispondere a specifiche esigenze del benchmark.

Parser AI e approccio basato sulla composizione La famiglia principale di strategie è quella basata sull'intelligenza artificiale. Invece di creare una classe distinta per ogni possibile combinazione di modello LLM e tecnica di pulizia del DOM, si è optato per un approccio basato sulla composizione tramite la classe `ConfigurableAiParser`.

Questa implementazione non contiene logiche di estrazione cablate, ma funge da orchestratore interno: al momento della costruzione, riceve in input un adapter AI (sezione 4.2.2) e l'identificativo di una specifica strategia di preprocessing. Quando

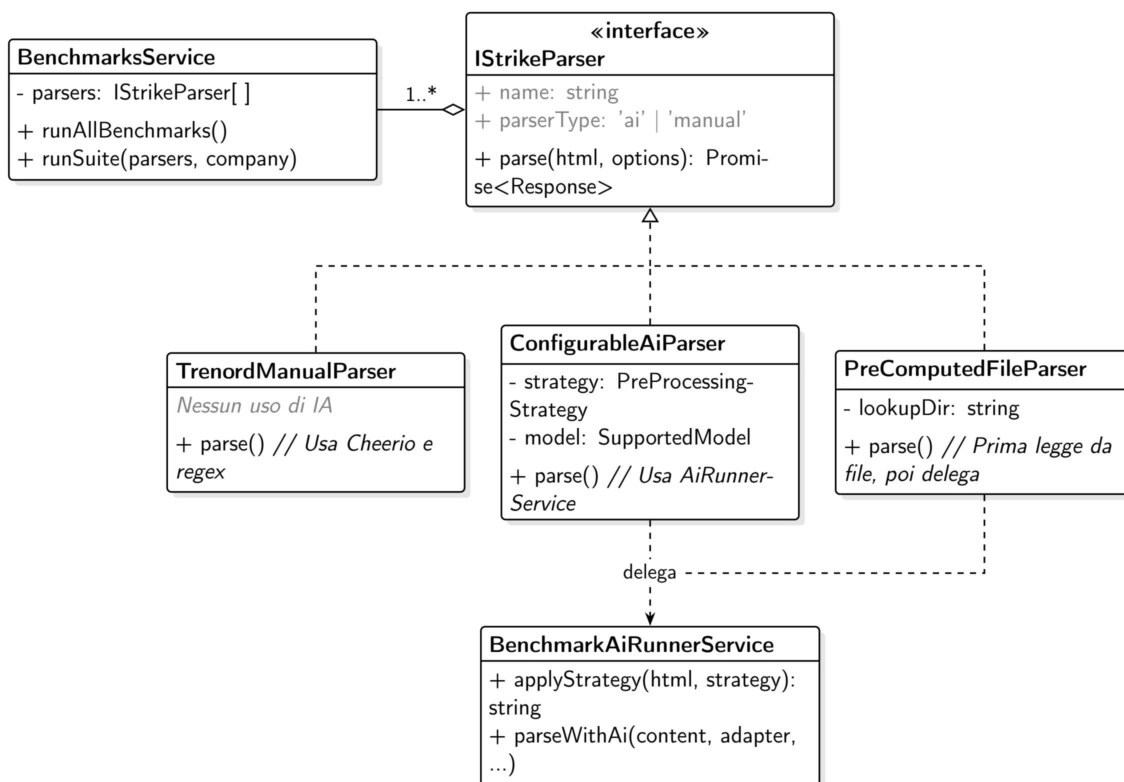

Figura 4.3: Diagramma delle classi del pattern strategy. Il *context* (`BenchmarksService`) orchestra l'estrazione iterando su un array di `IStrikeParser` agnostici. L'architettura evidenzia come la variante `ConfigurableAiParser` funga da ponte, delegando l'applicazione del preprocessing e l'invocazione dell'LLM al `BenchmarkAiRunnerService`.

viene invocato il metodo `parse`, la classe delega al `BenchmarkAiRunnerService` l'applicazione della pulizia del testo, la costruzione del prompt e l'invocazione del modello, restituendo poi il risultato normalizzato.

Questo design è il vero motore della tesi: permette di generare dinamicamente l'intera matrice di test (es. *gpt-4o + markdown*, *gemini + dom-minimizzato*, ecc.) senza moltiplicare inutilmente le classi. L'approccio si è rivelato facilmente estendibile anche per casi limite: ad esempio, per i file PDF di Trenitalia TPER, è bastato derivare una classe specializzata (`TrenitaliaTperAiParser`) che intercetta l'input per estrarne il testo prima di immetterlo nella medesima pipeline AI.

Parser manuali come baseline di resilienza Per valutare oggettivamente il vantaggio introdotto dall'intelligenza artificiale, è stato necessario implementare una baseline di riferimento. Sono state quindi realizzate strategie deterministiche (come `TrenordManualParser` e `EavManualParser`) che estraggono le informazioni navigando l'albero DOM ed espressioni regolari mirate su specifici selettori CSS (es. `.date-news`).

È importante precisare che questi script sono stati ottimizzati ad hoc per le rispettive fonti, con lo scopo di raggiungere un'accuratezza estrattiva prossima al 100% sui documenti originali. Questa scelta metodologica è voluta per simulare le prestazioni di un sistema tradizionale sviluppato su misura e perfettamente calibrato sulla struttura della pagina al momento dell'acquisizione.

Sebbene questi parser includano logiche di fallback testuale per mitigare le variazioni strutturali minori, essi incarnano deliberatamente la rigidità e la fragilità dei classici scraper web. Grazie al pattern strategy, queste implementazioni manuali possono essere iniettate nella suite di test in modo del tutto trasparente rispetto ai parser AI, rendendole il termine di paragone ideale per quantificare le «rotture» durante i test di manipolazione del codice descritti nella sezione 4.4.

Parser pre-calcolati per l'ottimizzazione dei costi Un'ultima categoria di implementazioni, rappresentata dal `PreComputedFileParser`, risolve un problema prettamente prestazionale. Come discusso nel capitolo 3, alcune strategie di preprocessing si affidano a Small Language Model esterni, come `jina-reader` o `mineru-html`. Eseguire l'inferenza di questi modelli in tempo reale per ogni iterazione della matrice di test avrebbe dilatato i tempi di esecuzione e generato costi di API insostenibili.

L'astrazione fornita dal pattern strategy ha permesso di aggirare l'ostacolo in modo elegante. Il preprocessing tramite questi modelli pesanti è stato eseguito *una tantum* sull'intero dataset, salvando i risultati su disco (in formato Markdown o HTML

minimizzato). Il `PreComputedFileParser` si limita a intercettare la richiesta di parsing, caricare il file pre-calcolato dal file system e passarlo al `ConfigurableAiParser` sottostante. In questo modo, il motore di benchmark rimane ignaro del fatto che la trasformazione sia avvenuta a priori, mantenendo intatta l'interfaccia e la fluidità dell'esecuzione.

4.2.4 La funzione `applyStrategy` per il preprocessing

A supporto del `ConfigurableAiParser`, la funzione `applyStrategy` traduce in codice le trasformazioni descritte nella sezione 3.3. L'implementazione è progettata per massimizzare il rapporto segnale-rumore e delegare le operazioni pesanti a librerie specializzate:

- **`basic-cleanup` e `dom-distillation`**: attraverso l'uso di selettori e iterazioni ricorsive, vengono eliminati i nodi non testuali e scartati gli attributi non essenziali, preservando la struttura gerarchica minima.
- **`html-to-markdown`**: esegue `basic-cleanup` per rimuovere il rumore, per poi convertire l'HTML pulito in Markdown.
- **Strategie SLM (`jina-reader` e `mineru-html`)**: a differenza delle precedenti, queste non prevedono elaborazione a runtime all'interno del modulo. La funzione agisce come un'interfaccia di I/O che carica i documenti pre-elaborati offline tramite i modelli esterni, come descritto nella sezione 3.3.

L'approccio modulare di questa funzione permette di alternare le strategie durante i benchmark tramite il semplice passaggio di una costante, garantendo che ogni modello riceva l'input nel formato esatto previsto dal disegno sperimentale.

4.3 Orchestrazione e pipeline AI

Il pattern adapter e il pattern strategy sono componenti chiave, ma per trasformare la definizione astratta degli esperimenti in una serie di esecuzioni concrete è necessario un livello di orchestrazione che coordini l'intero processo. Questo ruolo è svolto principalmente da due servizi all'interno del modulo benchmark:

- **`BenchmarksService`**: agisce da macro-orchestratore. È responsabile della generazione della matrice combinatoria di test, della gestione della concorrenza globale, dell'aggregazione statistica dei risultati e del salvataggio dei report finali.
- **`BenchmarkAiRunnerService`**: è il sub-orchestratore per i parser basati su intelligenza artificiale. Riceve in input il documento, applica la strategia di pre-

processing, calcola l'hash per il caching, interagisce con gli adapter e calcola i costi in base ai token consumati.

Insieme, questi due componenti trasformano la definizione astratta degli esperimenti in una serie di chiamate concrete alle API dei provider, raccogliendo in modo sistematico e riproducibile tutti i dati per le analisi.

La prima responsabilità del `BenchmarksService` è la generazione della matrice combinatoria di test, che definisce la configurazione logica del benchmark. Sfruttando le astrazioni fornite dai design pattern, il servizio calcola il prodotto cartesiano tra l'insieme dei modelli LLM da valutare, le strategie di preprocessing e i documenti grezzi del dataset.

Ogni coppia univoca risultante viene istanziata come un oggetto `ConfigurableAiParser`. Questo approccio permette di gestire centinaia di configurazioni sperimentali attraverso un'interfaccia di esecuzione uniforme, disaccoppiando la definizione del test dalla sua esecuzione materiale.

I parser così generati vengono organizzati in **suite** logiche suddivise per azienda di trasporto (es. Trenord, ATAC). Tale strutturazione facilita sia l'esecuzione selettiva di specifici segmenti del benchmark, sia la successiva aggregazione statistica dei risultati per dominio di provenienza. Nella fase finale di orchestrazione, il servizio incrocia iterativamente i parser di ogni suite con i relativi documenti del dataset, trasformando questa configurazione astratta in una sequenza di migliaia di task asincroni pronti per essere immessi nella coda di elaborazione.

Una volta costruita la lista dei task, il servizio deve eseguirli in modo efficiente, ma senza incorrere nei limiti imposti dai provider di LLM. I provider di modelli linguistici, infatti, impongono limiti stringenti sul numero di richieste al minuto (RPM) e sul numero di token al minuto (TPM). Superare questi limiti può portare a errori HTTP 429 (Too Many Requests) o addirittura a sospensioni temporanee dell'account API, con conseguente interruzione del benchmark e perdita di dati.

Per gestire questo problema, si crea un limiter con un numero massimo di esecuzioni concorrenti (intorno a 25 richieste parallele: nei benchmark effettuati, questo valore ha mostrato il miglior equilibrio tra velocità e prevenzione del rate limiting). Ogni task viene racchiuso in una funzione che, quando eseguita, avvia il parser corrispondente. Il limiter si assicura che non vengano mai avviate più di N funzioni contemporaneamente, accodando le successive fino al completamento di alcune di quelle in esecuzione.

Questo meccanismo, seppur semplice, si è rivelato sufficiente per rispettare i rate limit della maggior parte dei provider. Per quelli con limiti particolarmente stringenti, come Groq, si è reso necessario un ulteriore livello di controllo implementato

direttamente nell'adapter (come descritto nella sezione 4.2.2), ma il limiter globale fornisce una prima «linea di difesa».

È importante notare che il controllo della concorrenza non serve solo a evitare errori, ma anche a gestire i costi in modo prevedibile. Eseguire troppe richieste in parallelo può portare a un consumo molto rapido del budget giornaliero, rendendo difficile monitorare la spesa in tempo reale. Limitare la concorrenza permette di distribuire il carico in modo più uniforme e di avere un maggiore controllo sull'avanzamento degli esperimenti.

Va notato che alcuni provider offrono modalità di elaborazione asincrona per richieste di grandi dimensioni, a costo ridotto (fino al 50% in meno), ma con tempi di risposta che possono arrivare a 24 ore e, in alcuni casi, come per Groq, estendersi fino a sette giorni. Per le esigenze di questo studio, che richiedeva iterazioni rapide per affinare prompt e metriche, si è scelta la strada delle chiamate sincrone, sacrificando l'economicità in cambio della velocità di feedback. L'architettura ad adapter rende comunque agevole un'integrazione futura di questa modalità.

4.3.1 Implementazione degli schemi *strict* e *lenient*

Come discusso nel capitolo 3, una delle variabili sperimentali fondamentali riguarda il grado di rigidità dello schema di output richiesto all'LLM. Per supportare questa sperimentazione, sono state realizzate due distinte implementazioni degli schemi di validazione e una funzione di normalizzazione dedicata per il caso *lenient*.

Schema *strict* Lo schema *strict* rappresenta l'approccio più rigoroso: impone che l'output dell'LLM sia esattamente conforme alla struttura e ai vincoli definiti, senza alcuna tolleranza per deviazioni. La sua implementazione è la seguente:

```
export const StrikeDataSchema = z.object({
  startDate: z.string().regex(/^\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}$/),
  endDate: z.string().regex(/^\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}$/),
  locationType: z.enum(["REGIONAL", "NATIONAL"]),
  locationCodes: z.array(z.string()).optional(),
  guaranteedTimes: z.array(z.string()).optional()
});

export const BenchmarkStrikeSchema = z.discriminatedUnion("isStrike", [
  z.object({ isStrike: z.literal(false), reason: z.string() }),
  z.object({ isStrike: z.literal(true), strikeData: StrikeDataSchema })
]);
```

Questa implementazione impone validazione formale rigorosa: le date devono corrispondere esattamente alla regex specificata, `locationType` deve essere uno dei due valori dell'enum, la struttura dell'unione discriminata è tassativa. Qualsiasi deviazione, anche minima, causa il fallimento della validazione e l'intero oggetto viene scartato.

Questa implementazione viene utilizzata sia come ground truth (i file annotati manualmente seguono questo schema) sia, in alcune configurazioni di benchmark, come validatore diretto dell'output dell'LLM (modalità *strict*).

Nota sul formato della data Il formato scelto per le date non include intenzionalmente il fuso orario, a differenza di quanto prescrive lo standard ISO 8601 esteso. Questa decisione risponde a un'esigenza pratica emersa durante i test preliminari: i modelli linguistici tendono a confondersi quando devono interpretare e normalizzare informazioni relative a fusi orari, introducendo errori sistematici nella conversione di tempi tra zone diverse. Fornendo date prive di contesto di fuso orario, si elimina completamente questa fonte di confusione.

Schema *lenient* Lo schema *lenient* è stato progettato per essere più permissivo, spostando la responsabilità della normalizzazione formale dal modello al codice deterministico. La sua implementazione rilassa i vincoli tipografici e strutturali:

```
export const LenientStrikeDataSchema = z.object({
  isStrike: z.boolean(),
  reason: z.string().optional().nullable(),
  strikeData: z.object({
    startDate: z.string().describe("Data inizio formato yyyy-MM-dd
    ↪  HH:mm:ss"),
    endDate: z.string().describe("Data fine formato yyyy-MM-dd
    ↪  HH:mm:ss"),
    locationType: z.string().describe("Uno tra: REGIONAL, NATIONAL"),
    locationCodes: z.array(z.string()).optional().nullable(),
    guaranteedTimes: z.array(z.string()).optional().nullable()
  }).optional().nullable()
});
```

Le differenze salienti rispetto allo schema *strict* sono:

- I campi enumerati diventano stringhe libere, accompagnate da una descrizione testuale (tramite `.describe()`) che suggerisce il formato atteso senza imporlo.

- I campi opzionali sono dichiarati esplicitamente come `optional().nullable()`, per gestire sia l'assenza del campo che valori `null` espliciti.
- L'unione discriminata è stata semplificata in un'unica struttura con campi opzionali, più facile da gestire per modelli di piccole dimensioni.

Limitazioni degli structured output con OpenAI

Sebbene il framework offra un'astrazione uniforme, l'integrazione di OpenAI ha richiesto una soluzione alternativa. La funzionalità nativa di *structured output* (tramite `Responses API`), pur garantendo la conformità del JSON, non supporta costrutti di `zod` essenziali per lo schema *strict* originale, come espressioni regolari, unioni discriminate (usate per distinguere tra avvisi di sciopero e non) e campi opzionali nativi (obbligando a unioni esplicite con `null`).

Per aggirare queste limitazioni, è stata definita una variante semplificata dello schema (`BenchmarkAiOpenAISchema`) ad hoc per questo provider, che appiattisce le unioni ed elimina i vincoli regex. La discrepanza tra il formato richiesto dall'API e quello atteso dal ground truth viene risolta in modo trasparente dal pattern adapter: il metodo `generateWithResponsesAPI` interroga il modello con lo schema compatibile, effettua una pulizia programmatica della risposta (ad esempio rimuovendo le chiavi con valore `null`) e applica infine la validazione dello schema *strict* originale prima di restituire l'oggetto al motore di benchmark, pronto per le fasi di confronto e valutazione.

Riconciliazione semantica e normalizzazione a valle

L'adozione dello schema *lenient* demanda al codice deterministico l'onere di ricondurre i dati grezzi estratti dall'LLM ai vincoli rigorosi dello schema *strict*. Questa fase di normalizzazione agisce a tutti gli effetti come il modulo di *Transform* della pipeline Extract, Transform, Load (ETL), e agisce su tre direttrici principali

- La risoluzione temporale analizza le date tramite euristiche che riconoscono formati disomogenei (es. presenza o assenza della «T», date espresse in linguaggio naturale come «domani») e le converte nello standard atteso.
- La riconciliazione geografica mappa le stringhe descrittive (es. «Modena» o «Emilia-Romagna») sui corrispondenti codici ISTAT tramite dizionari di lookup. Se il modello omette la tipologia territoriale, questa viene dedotta logicamente dalla natura delle province identificate.
- L'uniformazione sintattica bonifica i campi vettoriali, come le fasce d'orario garantite, da spazi superflui e li normalizza (ad esempio `"6.00-9.00"` viene trasformato in `"06:00-09:00"`).

Solo al termine di queste trasformazioni l'oggetto risultante viene sottoposto alla validazione definitiva dello schema *strict*.

4.3.2 Costruzione dinamica del prompt

Mentre il `BenchmarksService` gestisce il flusso complessivo degli esperimenti, la vera e propria interazione con i modelli è delegata al `BenchmarkAiRunnerService`. Questo servizio NestJS costruisce il prompt dinamicamente prima di invocare l'adapter e calcolare l'hash per la cache. Come descritto nella sezione 3.6, le istruzioni al modello vengono assemblate anche in base alla fonte analizzata.

L'implementazione inietta il contesto geografico tramite una funzione dedicata (`getCompanyContext`) che inserisce nel prompt una breve descrizione dell'azienda (es. «ATAC è l'azienda di trasporto pubblico di Roma, Lazio (12)»), rendendo il prompt finale assai dettagliato. La struttura completa dei template di sistema è consultabile nell'appendice A.

Questo accorgimento si rende necessario per prevenire le allucinazioni estrinseche descritte nella sezione 2.4: modelli di grandi dimensioni hanno conoscenza (e capacità di «ragionamento») sufficiente per associare «ATAC» a «Lazio», ma modelli più piccoli potrebbero non averla e inventare dati geografici quando il testo non li cita esplicitamente.

Il prompt specifica poi condizioni precise per la decisione del campo booleano `isStrike`: si esplicita che documenti con revoche, differimenti, report di adesione passata o avvisi sul traffico in tempo reale devono generare output negativo. Questa logica risolve le ambiguità semantiche delle fonti disomogenee che presentano spesso informazioni non riguardanti nuovi scioperi, ma che potrebbero essere erroneamente interpretate come tali. Nel capitolo 5.2 verrà mostrato come queste regole abbiano ridotto i falsi positivi.

Il prompt, prima di fornire il documento pre-processato in input, si conclude con due esempi in formato JSON che illustrano il comportamento atteso sia quando lo sciopero è confermato, sia quando l'avviso viene scartato. Questo approccio trasferisce la logica di dominio nel contesto del modello, permettendo a modelli con meno parametri di compensare la loro minore capacità di ragionamento con direttive chiare.

4.3.3 Affidabilità, caching e raccolta dati

L'esecuzione di decine di migliaia di chiamate API richiede meccanismi adeguati per garantire la continuità dei test, il contenimento dei costi e la tracciabilità dei risultati. Per rispondere a queste esigenze, l'architettura del `BenchmarkAiRunnerService`

affianca al controllo della concorrenza un sistema di caching idempotente, una gestione degli errori basata su retry e una pipeline strutturata per l'aggregazione dei dati.

Prima di invocare una chiamata API, il servizio genera una firma univoca MD5 della richiesta, calcolata combinando il prompt completo (istruzioni, contesto e testo pre-processato), l'identificativo del modello, la strategia di preprocessing, il tipo di schema richiesto (lenient o strict) e i parametri di configurazione. Se nella directory locale `.ai_cache` esiste già un file corrispondente a questo hash, il sistema ne carica il contenuto omettendo la chiamata di rete.

Questo approccio offre diversi benefici: da un lato assicura la riproducibilità degli esperimenti, fornendo un archivio permanente delle risposte grezze utile anche per le analisi qualitative; dall'altro riduce i tempi di esecuzione e i costi delle API. Ad esempio, iterare sullo sviluppo dell'algoritmo di scoring o correggere bug non richiede di ripetere le chiamate fatturabili ai provider. La cache serializza inoltre la durata originale della chiamata (`durationMs`) e il conteggio dei token consumati, consentendo di calcolare le metriche di latenza e di costo anche in modalità offline.

Anche adottando queste precauzioni, le interazioni con i provider esterni rimangono soggette a potenziali interruzioni, come timeout di rete, errori temporanei dei server (HTTP 503) o il raggiungimento dei rate limit. Per rendere il benchmark più resiliente, è stato implementato un meccanismo di retry con *exponential backoff*: in caso di errore, il task viene ritentato con intervalli crescenti (ad esempio 1, 2, 4 fino a 32 secondi) per un massimo di cinque tentativi. Questa logica si coordina con il sistema di cache: se un task fallisce in modo definitivo, l'errore viene registrato ma non viene salvato alcun file fittizio. In questo modo, a una successiva esecuzione, il sistema tenterà nuovamente l'operazione, evitando che problemi temporanei causino perdite definitive di dati.

Man mano che i task vengono completati, il macro-orchestratore `BenchmarksService` ne accumula i risultati in memoria. Per ogni valutazione vengono registrati:

- L'identificativo del documento (fonte, data, URL).
- La configurazione del test (modello e strategia).
- L'oggetto strutturato estratto e l'esito della validazione formale.
- I metadati prestazionali ed economici (token, durata, costo stimato).
- I risultati dell'algoritmo di scoring (Precision, Recall, F1 e difformità di dettaglio), descritto nella sezione 4.4.2.

Al termine della matrice di test, i dati aggregati vengono serializzati in un report JSON completo e salvati nella directory `results/` includendo un timestamp esplicito e i dettagli della configurazione di lancio. Questo file costituisce la base dati per le successive visualizzazioni statistiche.

L'insieme di queste soluzioni tecniche ha consentito al framework di gestire la mole di test richiesta: nella sola campagna sperimentale di questa ricerca sono state eseguite oltre 30.000 chiamate API ai vari provider di LLM, mantenendo tempi di esecuzione contenuti (circa 10 minuti per una run di 2000 test). Sebbene per utilizzi futuri su ordini di grandezza superiori si possa valutare l'integrazione di una coda persistente (come BullMQ), per gli scopi attuali l'architettura basata su `p-limit` e file system si è rivelata adeguata. Una discussione sull'impatto economico effettivo di queste chiamate e sul rapporto costo-accuratezza è rimandata al capitolo 5.

4.4 Analisi della resilienza e metodologia di valutazione

In questa sezione vengono descritti due componenti realizzati per le analisi sperimentali. Il primo è il test di mutazione del DOM, uno specifico benchmark progettato per valutare la resilienza degli scraper manuali a fronte di modifiche strutturali della pagina. Il secondo è l'algoritmo di scoring utilizzato per quantificare l'accuratezza delle estrazioni.

4.4.1 Il generatore di «DOM chaos»

Per supportare i test di resilienza descritti nel capitolo 3, è stato implementato uno strumento dedicato: il generatore di «DOM chaos». Si tratta di una funzione pura che, ricevuto in input un documento HTML sotto forma di stringa, restituisce una nuova stringa HTML modificata secondo una serie di trasformazioni programmate, preservando integralmente il contenuto testuale visibile.

L'implementazione carica l'HTML in una struttura ad albero manipolabile programmaticamente. Le trasformazioni applicate sono le seguenti:

- Le classi CSS vengono rinominate: tutti i valori presenti negli attributi `class` sono sostituiti con stringhe casuali generate al momento (ad esempio, `date-news` diventa `xyz-123`). L'implementazione itera su tutti gli elementi con attributo `class` e applica una funzione di mapping che associa a ogni classe originale un identificatore univoco e casuale.
- I tag semantici con significato strutturale (come `<b>`, `<h4>`, `<li>`, `<ul>`, `<article>`) vengono rimpiazzati con tag generici `<div>` o `<span>`.

- A ogni nodo selezionato vengono iniettati attributi «rumore» casuali (es. `data-random="abc"`) che non alterano la visualizzazione ma aumentano la complessità del markup.
- Il testo viene manipolato individuando gli elementi che contengono solo testo (senza figli) e avvolgendo il loro contenuto in un tag `<span>` con classe fissa `wrapper-z`. Questa operazione «rompe» selettori come `div > text`.
- Vengono inseriti elementi fantasma: al DOM, in posizioni casuali, vengono aggiunti elementi vuoti o con testo fittizio (es. `<div class="hidden">contenuto non rilevante</div>`) per alterare gli indici posizionali.

Lo strumento è stato eseguito una sola volta sul dataset. Per ogni file, la funzione `messUpDom` ha prodotto una versione trasformata del documento. Questo dataset trasformato è stato poi utilizzato come input per i benchmark di resilienza, esattamente come si è fatto per le strategie basate su Jina Reader o MinerU: i file pre-calcolati sono stati letti dal disco e processati dai parser allo stesso modo dei file originali.

4.4.2 Algoritmo di scoring

La valutazione degli output generati dai parser rispetto al ground truth è delegata alla funzione dedicata `compareStrikes`.

Piuttosto che limitarsi a un confronto binario sull'intero documento, l'algoritmo adotta un approccio granulare: scompone l'oggetto JSON e valuta ogni campo dello schema come una distinta «unità informativa». Questo permette di misurare accuratamente le estrazioni parziali. La funzione restituisce un oggetto `ComparisonResult` contenente i contatori dei veri positivi (TP), falsi positivi (FP) e falsi negativi (FN), assieme alle metriche aggregate.

Le regole di confronto implementate variano in base alla tipologia di dato analizzato:

Gestione dei campi scalari e casi limite Per i campi a valore singolo (`isStrike`, `startDate`, `endDate`, `locationType`), il confronto applica alcune normalizzazioni per garantire robustezza:

- Le stringhe vengono troncate ai primi 16 caratteri (`yyyy-MM-dd HH:mm`). Questa euristica ignora discrepanze su secondi e millisecondi, irrilevanti ai fini informativi ma potenziale causa di falsi negativi formali.
- Se un campo opzionale è presente nel ground truth ma omesso dal modello, genera un **falso negativo**. Se inventato dal modello (assente nel ground truth), genera un **falso positivo**.

- Per quanto riguarda gli enum, il confronto su `locationType` è esatto e case-sensitive: l'onere di ricondurre varianti testuali come «REGIONALE» ai valori standard (ad esempio `"REGIONAL"`) è infatti rimandato a monte, alla funzione di normalizzazione descritta nella sezione 4.3.1.

Se il valore generato coincide con l'atteso, si incrementa il contatore TP. Una discrepanza su un campo scalare penalizza doppiamente: genera un FP (il modello ha inventato un dato errato) e un FN (non ha riconosciuto il dato corretto). Questa penalizzazione, per quanto severa, è coerente con l'errore logico che si commette.

Gestione dei campi vettoriali Il primo passaggio consiste nella deduplicazione degli array per impedire che un modello gonfi artificialmente il numero di TP ripetendo lo stesso valore corretto. Il calcolo delle metriche sui campi vettoriali sfrutta la logica insiemistica. Dati l'insieme dei valori generati dal modello (G) e l'insieme dei valori attesi dal ground truth (T), preventivamente deduplicati, i contatori vengono calcolati come segue:

- **Veri positivi (TP)**: cardinalità dell'intersezione ($|G \cap T|$).
- **Falsi positivi (FP)**: cardinalità della differenza ($|G \setminus T|$).
- **Falsi negativi (FN)**: cardinalità della differenza inversa ($|T \setminus G|$).

Così facendo, si penalizza la *recall* se il modello omette elementi validi e si penalizza la *precision* se aggiunge elementi allucinati.

Calcolo delle metriche aggregate e integrazione

Una volta elaborati tutti i campi, l'algoritmo dispone dei totali globali di TP, FP e FN per il record. Su questi valori vengono applicate le formule standard per il calcolo di **precision**, **recall** e **F1-score**. La funzione implementa i necessari controlli per le divisioni per zero (ad esempio, restituendo precisione 0 in assenza totale di estrazioni).

La tabella 4.2 riassume le regole logiche di assegnazione degli esiti per ogni tipologia di campo, con esempi concreti tratti dal dominio degli scioperi.

Il `BenchmarkAiRunnerService` invoca questa procedura per ogni test completato. I risultati quantitativi del `compareStrikes` vengono fusi con i metadati di esecuzione (costo, latenza in millisecondi, token consumati) e salvati nel report JSON finale, costituendo la base dati per le visualizzazioni statistiche analizzate nel capitolo successivo.

Esito	Criterio logico	Esempio di penalità nel dominio
Vero positivo (TP)	Il valore estratto corrisponde esattamente al ground truth.	Estrae correttamente la data "2026-03-27 09:00:00".
Falso positivo (FP)	Il modello «inventa» un dato assente nel ground truth.	Il testo dice «sciopero nazionale», ma il modello compila `locationCodes: ["12"]`.
Falso negativo (FN)	Il modello omette un dato presente nel ground truth.	Il testo indica «fascia di garanzia 06:00-09:00», ma il modello restituisce array vuoto.
Discrepanza scalare	Il modello estrae un valore scalare errato (es. una data sbagliata).	Genera **1 FP** (dato errato inserito) e **1 FN** (dato corretto omesso).

Tabella 4.2: Regole di assegnazione degli esiti nell'algoritmo di scoring per campi scalari e vettoriali.

4.4.3 Visualizzazione dei risultati

Per analizzare i risultati è stato realizzato uno script Python separato, `view_results.py`, che carica i file JSON prodotti dal benchmark in un `DataFrame` Pandas e genera grafici salvati come PNG nella directory `charts/`: F1-score medio per modello e strategia, costo vs accuratezza, tasso di allucinazione e confronto di resilienza tra dataset originale e alterato (*DOM chaos*). I grafici prodotti con `matplotlib` e `seaborn` verranno mostrati e analizzati nel capitolo 5.

Capitolo 5

Valutazione dei risultati: accuratezza, costi e resilienza strutturale

Questo capitolo presenta l'analisi empirica condotta attraverso **SWEET**, il framework descritto nel capitolo 4. L'obiettivo è fornire risposte quantitative alle domande di ricerca formulate nel capitolo 1, valutando l'uso degli LLM nel web scraping non solo in termini di accuratezza, ma anche di costi operativi e resilienza strutturale.

5.1 Metodologia di valutazione dell'accuratezza

Come discusso nella sezione 2.7, il mero confronto binario (successo/fallimento) dell'intero record estratto risulta inadeguato per valutare la complessità dell'estrazione semantica. Pertanto, la valutazione quantitativa dell'accuratezza in questo capitolo si basa sulle metriche di **precision, recall e F1-score**.

Il calcolo di tali metriche non avviene a livello di intero documento, ma in modo granulare per ciascun campo dello schema JSON. Come descritto in dettaglio nella sezione 4.4.2 (relativa all'implementazione dell'algoritmo di scoring), il motore di benchmark scompone l'output e assegna dinamicamente i veri positivi (TP), falsi positivi (FP) e falsi negativi (FN) tenendo conto della natura dei dati (distinguendo ad esempio tra stringhe esatte e insiemi parzialmente corretti negli array).

Oltre alle classiche metriche statistiche aggregate, l'analisi dei risultati farà ricorso a due indicatori specifici, utili per interpretare il comportamento e i limiti dei modelli linguistici:

- «Tasso di allucinazione»: definito matematicamente come $1 - \text{precision}$. Questa metrica misura la proporzione di elementi estratti che non corrispondono al ground truth, indicando quanto il modello «inventi» informazioni assenti nel testo originario. Un valore elevato segnala una pericolosa propensione del modello a introdurre dati sintatticamente validi ma semanticamente errati.
- «Phantom strike»: questo termine conia i casi in cui il modello rileva l'annuncio di un nuovo sciopero inesistente (generando un falso positivo sul campo discriminatorio `isStrike`). Questo tipo di errore logico è particolarmente grave in quanto non solo introduce dati errati, ma distorce completamente la comprensione del documento, portando la precision e l'F1-score a zero per l'intero record.

5.2 L'importanza del prompt engineering

Prima di poter confrontare l'efficienza delle diverse strategie di preprocessing o dei modelli, è stato necessario isolare e risolvere la principale fonte di errore emersa nelle fasi iniziali della sperimentazione: gli errori di classificazione logica (i «phantom strike»).

L'integrazione nel dataset di test degli avvisi storici dell'azienda ATAC (Roma) ha evidenziato in modo inequivocabile questo limite. A differenza di portali più strutturati, il sito di ATAC è caratterizzato da una forte ambiguità semantica: circa il 48% dei comunicati, pur essendo intitolati con la parola «sciopero», non annunciano nuovi scioperi, ma riportano revoche (es. «sospeso sciopero di domani»), informazioni sulla circolazione o dati statistici passati (es. «adesione del personale allo sciopero al 12%»).

5.2.1 Conseguenze di un prompt generico

Nella prima iterazione del benchmark, il sistema utilizzava un prompt generico (si veda l'appendice A per il testo completo). Di fronte all'ambiguità del dataset ATAC, i risultati sono stati deludenti.

Come si evince dalla figura 5.1, quasi tutti i modelli – specialmente quelli di dimensioni più contenute come Llama 4 Scout 17B – hanno registrato tassi di *phantom strike* prossimi al 46%. Leggendo la parola «sciopero» e individuando degli orari nel testo (pur riferiti al passato), i modelli tendevano a forzare l'estrazione impostando il campo `isStrike: true`, inventando di sana pianta le date o ricopiando quelle di eventi ormai conclusi.

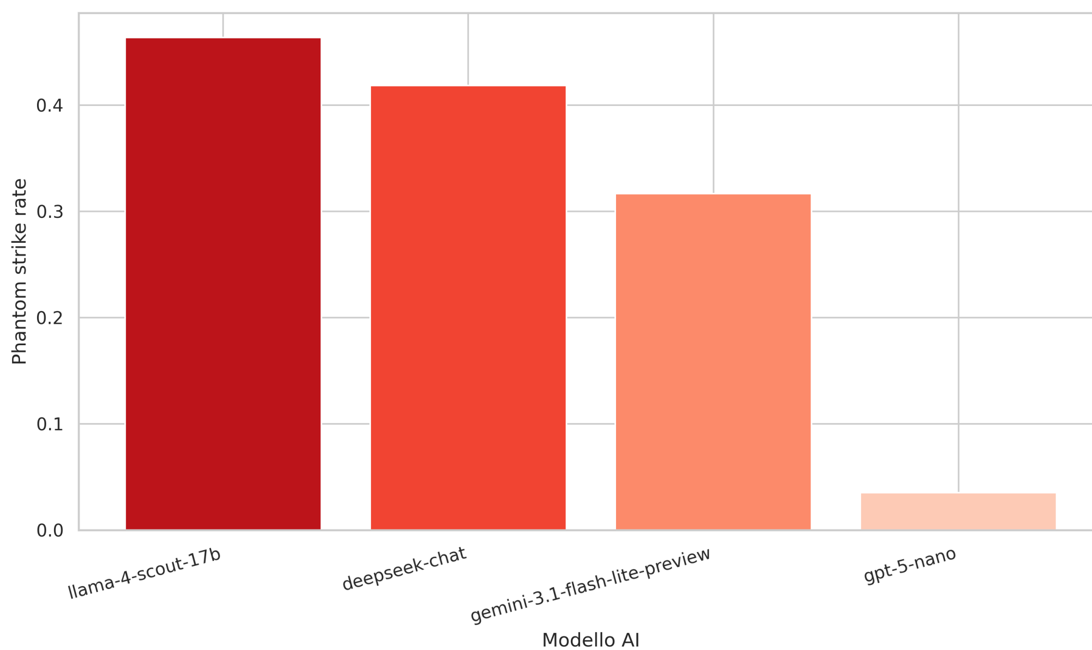

Figura 5.1: Tasso di phantom strike utilizzando un prompt generico (valori inferiori indicano maggiore precisione). I risultati per ciascun modello sono aggregati sulle varie strategie di preprocessing. Ad eccezione di GPT-5, tutti i modelli hanno fallito la classificazione in una percentuale rilevante di casi, che oscilla tra circa un terzo e quasi la metà degli stessi, scambiando revoche o report di adesione per nuovi scioperi.

L'unica eccezione è GPT-5-nano (errore sul 3,5%). Questa performance non deriva da capacità superiori del modello, ma dall'implementazione dello schema: come discusso nella sezione 4.3.1, tra le limitazioni degli structured output di OpenAI vi è che i campi opzionali non possono essere rappresentati nativamente, ma solo tramite unione con `null`. Questo ha richiesto descrizioni più esplicite dei campi, specialmente per `strikeData`, che viene istruito di essere `null` se l'avviso non tratta di un nuovo sciopero. Queste esplicitazioni hanno funzionato come istruzioni aggiuntive, rendendo il prompt per GPT-5-nano più informativo e adatto alla classificazione.

Questi risultati confermano quanto discusso nel capitolo 1: i modelli hanno seguito l'istruzione generica di «estrarre i dettagli dello sciopero» interpretando la mera presenza della parola chiave come condizione sufficiente, indipendentemente dal contesto effettivo. Pur non necessitando delle complesse euristiche o delle regole di classificazione manuali tipiche di un parser tradizionale, l'LLM ha dimostrato che la sola capacità di estrazione semantica non è sufficiente in assenza di una chiara logica di dominio per discriminare tra veri annunci, revoche e report.

Si è quindi passati a un prompt esplicito con regole di dominio (es. «se il testo è una revoca, imposta `isStrike` a `false`») ed esempi concreti di output negativo (tecnica di *few-shot prompting*), che ha azzerato i casi di phantom strike (allucinazione intrinseca) su tutti i modelli testati.

Il prompt ha inoltre svolto un ruolo importante nel contenimento delle allucinazioni estrinseche, ovvero la generazione di dati assenti nel testo. Come anticipato nella sezione 4.3.2, l'assenza di riferimenti geografici espliciti in alcune fonti (ad esempio, gli avvisi di ATAC che omettono di specificare «Roma» o «Lazio») spingeva i modelli più piccoli a inventare codici territoriali errati o classificare lo sciopero come «nazionale». L'iniezione programmatica di un contesto geografico, anche se sintetico, ha fornito ai modelli le informazioni necessarie per classificare molto più accuratamente le posizioni geografiche.

Come si osserva nella figura 5.2, l'utilizzo del prompt ottimizzato ha permesso a tutti i modelli di raggiungere livelli di accuratezza elevati. Un dato di particolare interesse è che modelli di dimensioni contenute, come `gpt-5-nano` e `gemini-3.1-flash-lite-preview` (i più piccoli ed economici delle rispettive famiglie), hanno ottenuto risultati paragonabili, se non talvolta superiori, a quelli dei modelli più grandi, assestandosi su un F1-score medio prossimo allo 0,95.

5.2.2 Complessità di estrazione per tipologia di campo

Sebbene l'uso del prompt esplicito abbia permesso di raggiungere un F1-score medio molto elevato, l'algoritmo di scoring granulare ha evidenziato come gli errori residui

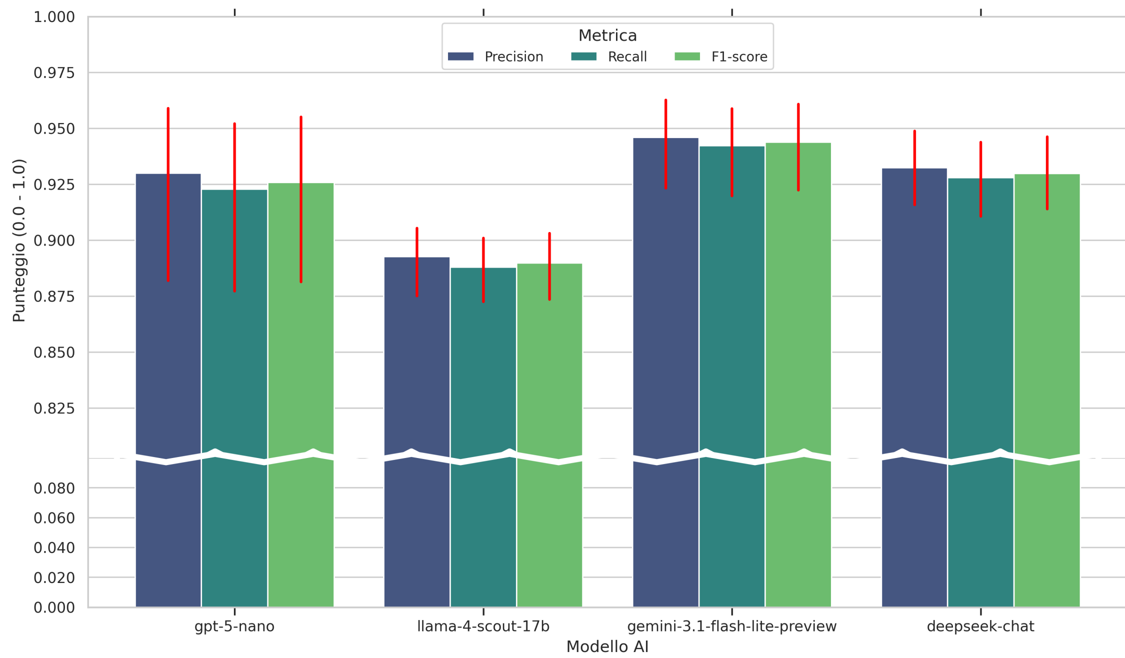

Figura 5.2: F1-score medio per modello utilizzando il prompt esplicito. Si nota come modelli più leggeri ed economici (gpt-5-nano, gemini-3.1-flash-lite-preview) mantengano prestazioni ampiamente superiori allo 0,90, rendendo l'estrazione praticabile a costi contenuti.

non siano distribuiti uniformemente. Non tutte le informazioni, infatti, presentano lo stesso livello di difficoltà di estrazione per un modello linguistico.

La tabella 5.1 mostra la distribuzione degli errori per singolo campo in una tipica esecuzione di benchmark con prompt ottimizzato.

Campo JSON	**Errori totali**	**Tasso di fallimento (%)**
`guaranteedTimes`	440	11,62%
`locationType`	335	8,84%
`endDate`	178	4,70%
`startDate`	79	2,09%
`isStrike`	51	1,35%
`locationCodes`	36	0,95%

Tabella 5.1: Distribuzione degli errori per singolo campo. Emerge una netta differenza tra informazioni complesse da estrarre e formattare sintatticamente (`guaranteedTimes`) e campi supportati dal contesto iniettato a monte (`locationCodes`).

I risultati confermano l'efficacia delle contromisure adottate nel prompt. Il campo `locationCodes` risulta quasi perfetto (errore inferiore all'1%): questo risultato eccellente è il frutto diretto dell'iniezione programmatica del contesto geografico. Fornendo a monte l'associazione tra l'azienda e il suo territorio di competenza, il modello non deve più «indovinare» o possedere conoscenza pregressa, abbattendo le allucinazioni estrinseche. Analogamente, il basso tasso di fallimento sul campo discriminatorio `isStrike` (1,35%) dimostra che le regole esplicite fornite in linguaggio naturale hanno ampiamente risolto il problema dei falsi positivi.

Le discrepanze maggiori emergono invece su altri fronti. La data di fine (`endDate`) presenta un tasso di errore più del doppio rispetto a quella di inizio (`startDate`). L'analisi qualitativa rivela che questo accade poiché la fine dello sciopero spesso non è esplicitata direttamente nel testo (es. «sciopero di 4 ore dalle 09:00»), imponendo al modello una deduzione logico-matematica oltre alla semplice estrazione.

Il campo con il tasso di fallimento più alto in assoluto è `guaranteedTimes` (11,62%). Questo si spiega con l'eccessivo sforzo richiesto all'LLM: da un lato l'identificazione semantica di fasce orarie che si trovano spesso immerse in un testo discorsivo complesso (es. «dall'inizio del servizio alle 8:30 e dalle 17:00 a fine servizio») o frammentate in tabelle di difficile lettura; dall'altro, il vincolo sintattico rigoroso imposto dallo schema, che pretende la traduzione di tale testo in un array di stringhe formattate come `HH:mm-HH:mm`.

5.3 Impatto del preprocessing e costi

Stabilita l'accuratezza del prompt, il primo quesito di ricerca indaga il trade-off tra compressione del documento HTML grezzo e perdita di contesto. Inviare un intero DOM non filtrato a un LLM, oltre ad esporre il modello a rumore sintattico, si traduce in un costo per richiesta che diventa insostenibile su larga scala.

5.3.1 Compressione per strategie di preprocessing

I test condotti su centinaia di avvisi reali hanno misurato l'efficacia delle diverse strategie di preprocessing in termini di riduzione del carico informativo. Partendo da una media di **271 KB** (circa **69.000 token**) per i file HTML grezzi, si sono registrati i seguenti ratei di compressione (rappresentati in figura 5.3):

- `basic-cleanup`: riduce il peso a circa 39 KB (9.970 token in media), con una compressione dell'85,69%. Eliminando script e stili, questa strategia preserva l'integrità del testo ma mantiene gran parte della verbosità sintattica dell'HTML.
- `jina-reader`: il modello SLM restituisce un Markdown pulito di circa 35 KB (7.060 token), raggiungendo una riduzione dell'89,86%. Sebbene efficace nel preservare le tabelle, tende a conservare elementi di navigazione non pertinenti.
- `html-to-markdown`: la conversione euristica standard riduce l'input a 23 KB (5.920 token), con una compressione del 91,50%. Si conferma un ottimo compromesso tra leggibilità per il modello e risparmio di token.
- `mineru-html`: l'approccio SLM aggressivo comprime la pagina a soli 2,7 KB (590 token), superando il 99% di compressione. La riduzione di oltre due ordini di grandezza nei token di input abbassa drasticamente i costi, a fronte però dei rischi di perdita di contesto analizzati nella sezione seguente.
- `dom-distillation`: isolando esclusivamente i nodi contenenti l'articolo, si scende a 1,6 KB (430 token), con una riduzione del 99,39%.
- `dom-distillation-markdown`: combinando la distillazione mirata con la conversione in Markdown, si ottiene l'input più denso e compatto in assoluto: appena 280 token medi per documento (una riduzione del 99,59% rispetto all'originale).

5.3.2 Impatto sull'accuratezza

La riduzione dei token si traduce in un risparmio economico considerevole, ma a quale impatto sulla qualità dell'estrazione? L'analisi aggregata dei risultati rivela

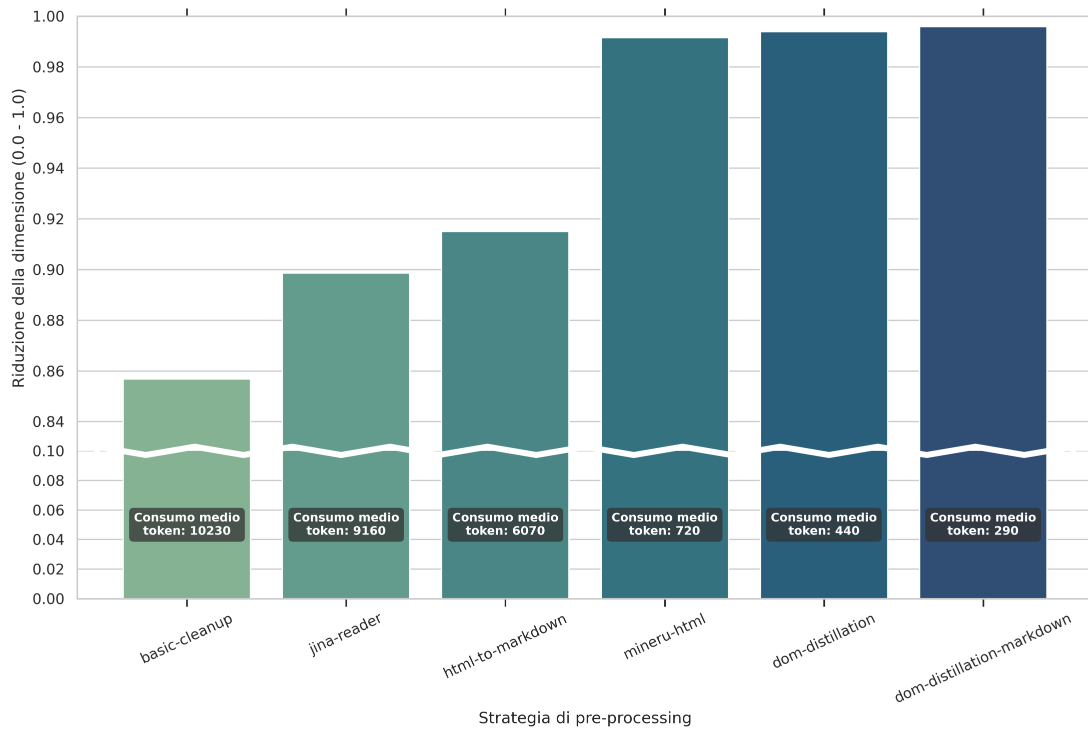

Figura 5.3: Riduzione percentuale della dimensione del contesto in input e consumo medio di token per strategia. Si nota come le strategie mirate riducano l'input di oltre 200 volte rispetto al documento grezzo.

che non esiste una proporzionalità lineare tra la pulizia dell'input e il miglioramento delle performance: l'efficacia dipende strettamente da *cosa* viene rimosso.

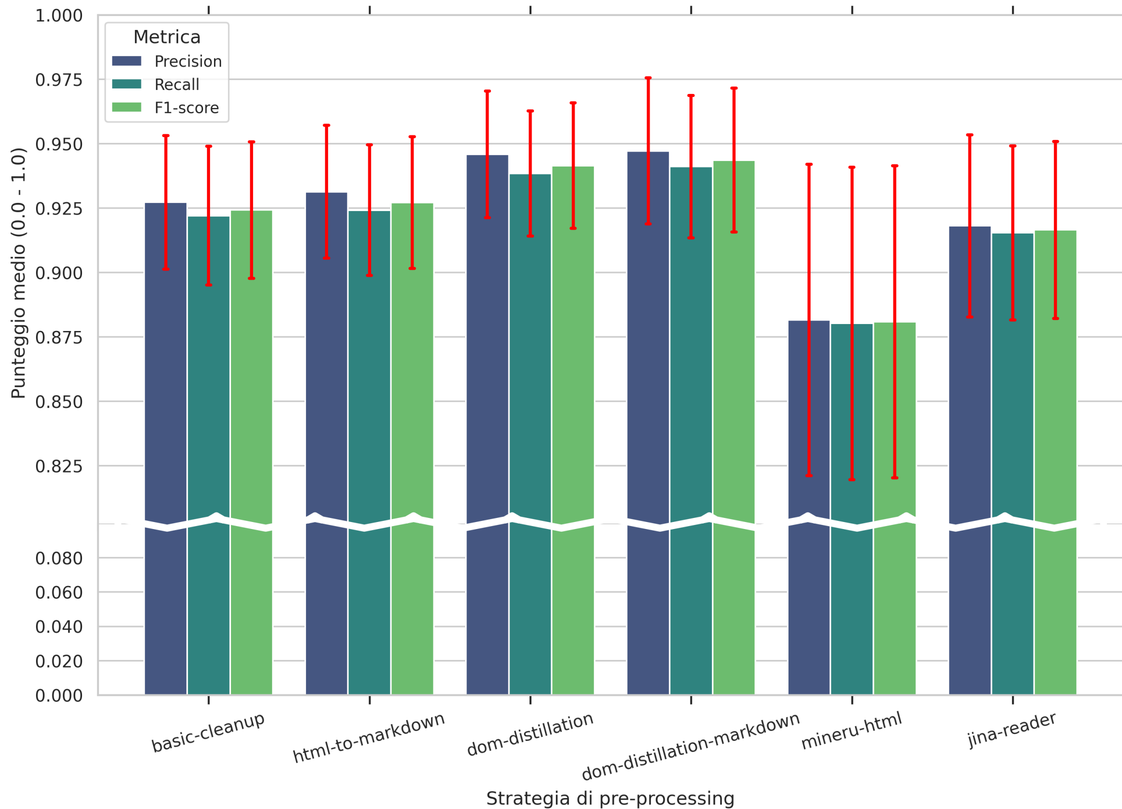

Figura 5.4: F1-score medio per strategia di preprocessing. La strategia più mirata, `dom-distillation-markdown`, ottiene l'F1-score più elevato. Le strategie basate su semplici euristiche, come `html-to-markdown`, mostrano risultati modesti ma comunque superiori all'SLM di `mineru-html`, che presenta invece una prestazione mediamente inferiore e caratterizzata da forte varianza: se in alcuni casi raggiunge un F1-score $> 0{,}93$ rimuovendo efficacemente il solo rumore, in altrettanti una pulizia troppo aggressiva elimina anche contenuto semanticamente rilevante, evidenziando l'equilibrio delicato tra compressione e preservazione del contesto.

Le strategie di pulizia euristica, come `basic-cleanup` e `html-to-markdown`, mantengono i livelli di accuratezza più alti tra gli approcci non ibridi (con un F1 medio tra 0,91 e 0,93). Rimuovendo esclusivamente il rumore sintattico (script e stili), l'intero albero informativo viene preservato. La conversione in Markdown, in particolare, si rivela un formato intermedio che molti LLM riescono a interpretare nativamente con alta precisione.

Riguardo le strategie basate su Small Language Model dedicati al preprocessing, il servizio `jina-reader` offre prestazioni in linea con le pulizie euristiche (F1 $\approx$ 0,91), convertendo efficacemente l'HTML in Markdown, ma tendendo a conservare

elementi di *boilerplate* (come menu e footer) che aumentano il conteggio dei token rispetto a conversioni più mirate. Contrariamente, la strategia `mineru-html` paga la sua elevata compressione (99,1%) con un notevole degrado dell'accuratezza (l'F1 scende intorno a 0,88-0,89). L'analisi qualitativa degli errori offre una spiegazione del fenomeno: l'estrazione aggressiva del contenuto operata da questo SLM elimina sistematicamente metadati invisibili all'utente ma indispensabili alla comprensione del contesto da parte dell'LLM. Un caso emblematico riscontrato nei test è la perdita dell'anno solare: spesso l'anno di uno sciopero non è scritto nel corpo dell'avviso (che recita banalmente «sciopero venerdì 27 marzo»), ma è collocato nel tag `<title>` o nei meta-tag Open Graph. Privato di questo contesto globale, l'LLM è costretto a dedurre l'anno, sfociando frequentemente in allucinazioni estrinseche (inventando date errate) e causando il fallimento della validazione.

L'approccio ibrido `dom-distillation` emerge come la soluzione ottimale per siti web a struttura nota. Fornendo all'LLM esclusivamente il sotto-albero DOM contenente l'articolo (assicurandosi di concatenare programmaticamente elementi essenziali come la data di pubblicazione originale), si ottiene la massima compressione (99,3%) unita alla massima accuratezza (F1 costantemente superiore a 0,95 con modelli come DeepSeek-Chat e GPT-5-nano). L'assenza totale di footer, menu laterali e link ad articoli correlati elimina alla radice la possibilità che il modello venga «distratto» da scioperi passati citati altrove nella pagina, azzerando di fatto i casi di phantom strike.

5.3.3 Analisi costo-efficacia

Per valutare la fattibilità operativa delle diverse architetture, è necessario incrociare le metriche di accuratezza con i costi di esecuzione. Per rendere la metrica economica più intuitiva, i dati sono stati proiettati su un asse delle ascisse che rappresenta il numero di file processabili con 1 dollaro di spesa. In questa configurazione, le combinazioni ottimali (alta accuratezza e alta efficienza economica) si collocano nel quadrante in alto a destra dei grafici di dispersione.

Al fine di facilitare la leggibilità, l'analisi è stata suddivisa in due visualizzazioni: la prima relativa alle strategie di pulizia meno aggressive (euristiche e Jina Reader), la seconda relativa alle strategie ad alta compressione (DOM distillation e MinerU).

Il grafico in figura 5.5 illustra le performance sui contesti più lunghi. In questo scenario, `gpt-5-nano` ottiene i risultati di accuratezza più alti, superando lo 0,96 di F1-score quando abbinato a `basic-cleanup`. Per quanto riguarda `gemini-3.1-flash-lite-preview`, è opportuno precisare che i test sono stati eseguiti impostando il parametro «thinking level» al valore minimo (`MINIMAL`). Sebbene un livello di ragionamento superiore avrebbe plausibilmente incrementato l'F1-score, la scelta è

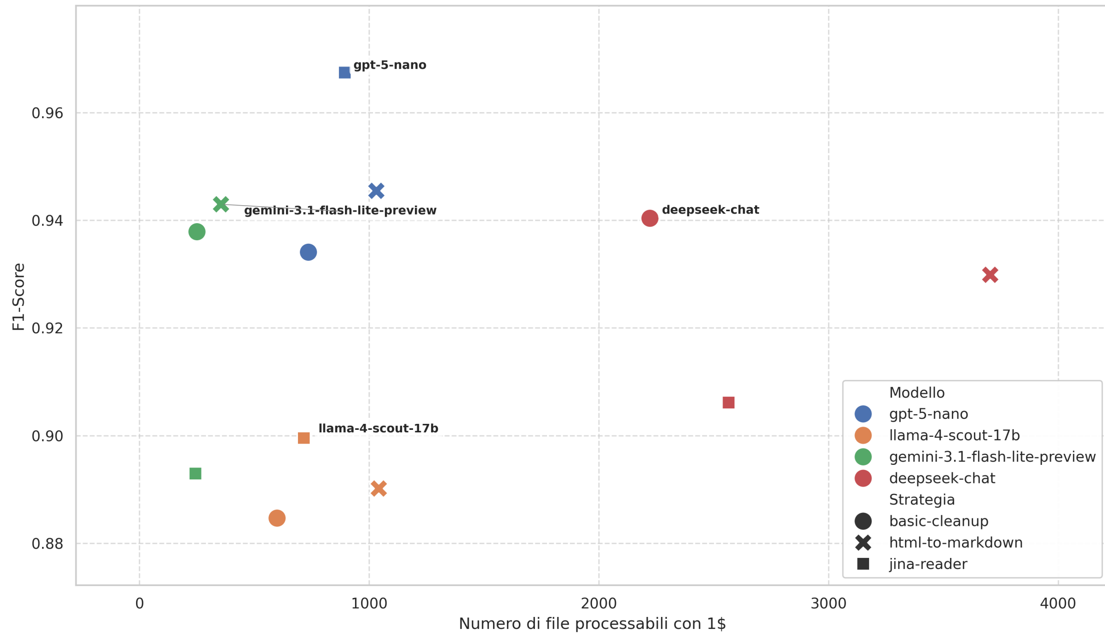

Figura 5.5: Dispersione dell'efficienza economica rispetto all'F1-score per strategie di preprocessing conservative (`basic-cleanup`, `html-to-markdown`, `jina-reader`).

stata dettata dalla necessità di contenere i costi: il modello si posiziona già nella fascia meno economica del grafico (elaborando circa 300 file per dollaro), e un aumento dei token di *reasoning* ne avrebbe ulteriormente ridotto l'efficienza economica rispetto alle alternative. Un dato rilevante emerge osservando `deepseek-chat`: pur registrando un F1-score leggermente inferiore (intorno a 0,93-0,94), si posiziona nettamente più a destra nell'asse, permettendo di processare tra i 2000 e i 4000 file con un dollaro. Inoltre, si nota come la transizione da `basic-cleanup` a `html-to-markdown` (rappresentata dalle «x» nel grafico) spinga generalmente i punti verso destra, confermando l'utilità del Markdown per ottimizzare la spesa mantenendo stabile l'accuratezza.

Il grafico in figura 5.6 mostra i risultati sulle strategie con una compressione più estrema. In questo contesto, l'approccio ibrido `dom-distillation-markdown` (rappresentato dai cerchi) si conferma la soluzione più accurata, permettendo a `gemini-3.1-flash-lite-preview` e `gpt-5-nano` di superare la soglia dello 0,95 di F1-score. La strategia `mineru-html` (le «x»), pur essendo altamente compressa, penalizza l'accuratezza di tutti i modelli testati, riportando l'F1-score su valori compresi tra 0,81 e 0,90, un comportamento in linea con la perdita di informazioni contestuali analizzata in precedenza.

L'evidenza più interessante di questa seconda visualizzazione riguarda il divario di

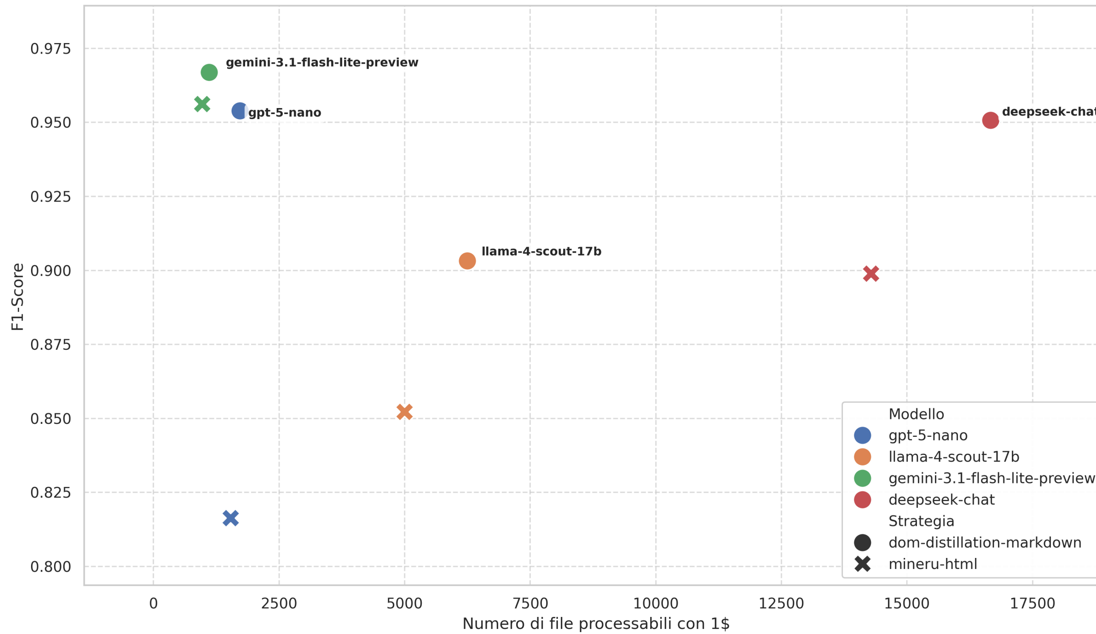

Figura 5.6: Dispersione dell'efficienza economica rispetto all'F1-score per strategie ad alta compressione (`dom-distillation-markdown`, `mineru-html`).

costo. L'abbinamento tra `deepseek-chat` e `dom-distillation-markdown` raggiunge un livello di efficienza notevole: il modello si attesta su un F1-score di 0,95, un valore solo marginalmente inferiore ai modelli di punta, ma permette di elaborare oltre 16.000 documenti per singolo dollaro. A titolo di confronto, a parità di strategia, `gemini-3.1-flash-lite-preview` elabora circa 1.000 file per dollaro.

Mentre le figure precedenti hanno illustrato i trend aggregati per singola dimensione (modello o strategia), le tabelle 5.2 e 5.3 presentano i risultati analitici per ogni singola combinazione testata nel framework. La tabella 5.2 sintetizza i valori numerici delle combinazioni più performanti emerse dall'analisi, includendo come benchmark di riferimento il parser manuale tradizionale. I risultati confermano quanto osservato nei grafici di dispersione: `DeepSeek-V3.2` con `dom-distillation-md` rappresenta il miglior compromesso costo-efficacia, mentre `GPT-5 nano` mantiene l'accuratezza più elevata a fronte di un costo superiore. La tabella 5.3 riporta nel dettaglio i costi per file e i tassi di allucinazione e phantom strike per tutte le configurazioni, evidenziando come l'efficienza economica si accompagni talvolta a un aumento degli errori.

Modello / parser	Preprocessing	F1-score	Precision	Recall
Scraper tradizionali				
Trenord manuale (regex)	-	0,9861	0,9861	0,9861
EAV manuale (regex)	-	0,9723	0,9792	0,9666
Estrazione semantica tramite LLM				
gpt-5-nano	jina-reader	0,9675	0,9704	0,9655
gemini-3.1-flash-lite	dom-distillation-md	0,9669	0,9709	0,9642
gemini-3.1-flash-lite	dom-distillation	0,9660	0,9695	0,9636
gemini-3.1-flash-lite	mineru-html	0,9562	0,9562	0,9562
gpt-5-nano	dom-distillation-md	0,9539	0,9587	0,9505
deepseek-chat	dom-distillation	0,9530	0,9568	0,9502
deepseek-chat	dom-distillation-md	0,9507	0,9531	0,9489
gpt-5-nano	html-to-markdown	0,9455	0,9510	0,9417
gemini-3.1-flash-lite	html-to-markdown	0,9430	0,9465	0,9405
deepseek-chat	basic-cleanup	0,9404	0,9429	0,9387
gemini-3.1-flash-lite	basic-cleanup	0,9379	0,9395	0,9367
gpt-5-nano	dom-distillation	0,9376	0,9449	0,9325
gpt-5-nano	basic-cleanup	0,9341	0,9383	0,9310
deepseek-chat	html-to-markdown	0,9299	0,9337	0,9273
llama-4-scout-17b	dom-distillation	0,9094	0,9122	0,9075
deepseek-chat	jina-reader	0,9062	0,9074	0,9054
llama-4-scout-17b	dom-distillation-md	0,9032	0,9063	0,9010
llama-4-scout-17b	jina-reader	0,8996	0,9012	0,8982
deepseek-chat	mineru-html	0,8989	0,9006	0,8975
gemini-3.1-flash-lite	jina-reader	0,8930	0,8935	0,8926
llama-4-scout-17b	html-to-markdown	0,8902	0,8944	0,8875
llama-4-scout-17b	basic-cleanup	0,8847	0,8885	0,8820
llama-4-scout-17b	mineru-html	0,8522	0,8532	0,8514
gpt-5-nano	mineru-html	0,8163	0,8166	0,8160

Tabella 5.2: Risultati quantitativi aggregati delle metriche di accuratezza (F1, Precision, Recall) ordinati per F1-score decrescente. La strategia di `dom-distillation-markdown` è indicata come «`dom-distillation-md`» per motivi di spazio. I modelli più performanti si attestano su F1-score superiori a 0,95, avvicinandosi ai risultati dei parser manuali.

Modello / parser	**Preprocessing**	**Costo ($ / file)**	**Halluci-nation rate (%)**	**Phantom strike rate (%)**
Scraper tradizionali				
Trenord manuale (regex)	-	n/d	n/d	n/d
EAV manuale (regex)	-	n/d	n/d	n/d
Estrazione semantica tramite LLM				
gpt-5-nano	jina-reader	0,0011	2,96	0,00
gemini-3.1-flash-lite	dom-distillation-md	0,0009	2,91	0,43
gemini-3.1-flash-lite	dom-distillation	0,0010	3,05	0,43
gemini-3.1-flash-lite	mineru-html	0,0010	4,38	0,00
gpt-5-nano	dom-distillation-md	0,0006	4,13	0,00
deepseek-chat	dom-distillation	0,0001	4,32	0,85
deepseek-chat	dom-distillation-md	0,0001	4,69	0,43
gpt-5-nano	html-to-markdown	0,0010	4,90	0,00
gemini-3.1-flash-lite	html-to-markdown	0,0028	5,35	0,43
deepseek-chat	basic-cleanup	0,0004	5,71	0,41
gemini-3.1-flash-lite	basic-cleanup	0,0040	6,05	0,41
gpt-5-nano	dom-distillation	0,0006	5,51	0,85
gpt-5-nano	basic-cleanup	0,0014	6,17	0,82
deepseek-chat	html-to-markdown	0,0003	6,63	0,00
llama-4-scout-17b	dom-distillation	0,0002	8,78	0,00
deepseek-chat	jina-reader	0,0004	9,26	0,68
llama-4-scout-17b	dom-distillation-md	0,0002	9,37	0,00
llama-4-scout-17b	jina-reader	0,0014	9,88	0,68
deepseek-chat	mineru-html	0,0001	9,94	0,62
gemini-3.1-flash-lite	jina-reader	0,0041	10,65	0,68
llama-4-scout-17b	html-to-markdown	0,0010	10,56	0,00
llama-4-scout-17b	basic-cleanup	0,0017	11,15	0,00
llama-4-scout-17b	mineru-html	0,0002	14,68	0,62
gpt-5-nano	mineru-html	0,0006	18,34	0,00

Tabella 5.3: Analisi dei costi per file e dei tassi di errore (allucinazioni e phantom strike). Le righe mantengono lo stesso ordinamento per F1-score della tabella 5.2.

5.4 Analisi della latenza e dei tempi di esecuzione

Oltre all'accuratezza e all'efficienza economica, l'implementazione di una pipeline di scraping con LLM deve fare i conti con un'ulteriore metrica: la latenza. Mentre i parser tradizionali basati su selettori CSS o espressioni regolari risolvono l'estrazione in pochi millisecondi, l'invocazione di un modello generativo introduce tempi di attesa ordini di grandezza superiori.

La figura 5.7 illustra la latenza media registrata per ciascun modello durante i benchmark. I dati evidenziano una variabilità considerevole.

Figura 5.7: Latenza media di esecuzione (in secondi) suddivisa per modello e strategia di preprocessing. Si nota come, a parità di modello, l'aumento della dimensione del contesto (es. `basic-cleanup`) dilati i tempi di risposta. Llama-4-Scout registra i tempi inferiori (scendendo sotto il secondo con strategie ad alta compressione), mentre GPT-5-nano supera costantemente i 15 secondi.

All'estremo inferiore dello spettro si posiziona `llama-4-scout`, che registra un tempo di risposta medio di soli 2,4 secondi. Come discusso teoricamente nella sezione 2.6, questo è dovuto all'hardware specializzato (architettura TSP) usato da Groq, sul quale il modello è ospitato. Questa architettura, progettata per massimizzare la velocità su richieste singole, consente di abbassare considerevolmente la latenza media.

Nella fascia intermedia si collocano `gemini-3.1-flash-lite-preview` e `deepseek-chat`, con latenze che oscillano tra i 3 e i 6 secondi. Tali tempistiche, pur non eguagliando l'istantaneità dei parser manuali, risultano del tutto compatibili con task di estrazione asincroni o pipeline ETL schedulate.

Il dato più critico riguarda invece `gpt-5-nano`. Nonostante questo modello abbia registrato i picchi di accuratezza più alti (F1 $\approx$ 0,96), i tempi di esecuzione si sono rivelati particolarmente dilatati, sfiorando in alcuni casi i 20 secondi per singolo documento. Tali latenze rendono questo modello inadatto a scenari di elaborazione *real-time* (ad esempio, un bot che deve rispondere istantaneamente all'utente).

L'analisi incrociata tra latenza e strategie di preprocessing ha confermato un principio architetturale atteso: ridurre il numero di token in input accelera l'elaborazione. Strategie ad alta compressione come `dom-distillation` o `mineru-html` riducono notevolmente il tempo necessario al modello per processare il prompt, con un impatto positivo sulla latenza complessiva.

Tuttavia, è interessante notare un'apparente anomalia prestazionale legata alla strategia `jina-reader`: sebbene produca un input con un numero medio di token superiore rispetto alla banale conversione euristica `html-to-markdown`, fa registrare

tempi di esecuzione complessivamente inferiori rispetto a quest'ultima. Come discusso teoricamente nella sezione 2.3, questo comportamento è spiegabile dalla natura del Markdown generato, specificamente addestrato per essere «LLM-friendly». L'assenza di artefatti sintattici e la maggiore coerenza strutturale riducono l'ambiguità del contesto, agevolando la fase di elaborazione del prompt iniziale e accelerando, di conseguenza, il processo di inferenza del modello a valle.

5.5 Carico cognitivo nello schema di output

Come definito nella metodologia (sezione 3.5) e illustrato a livello implementativo (sezione 4.3.1), il framework ha messo a confronto le due filosofie di validazione dell'output: *strict* e *lenient*. L'obiettivo di questa analisi è quantificare quanto il «carico cognitivo» imposto all'LLM – ovvero la richiesta di eseguire simultaneamente l'estrazione semantica e la formattazione sintattica rigorosa – incida sull'efficacia del processo, specialmente nei modelli di dimensioni ridotte.

5.5.1 Accuratezza complessiva e precisione

I risultati empirici dimostrano che alleggerire il modello dalla responsabilità della formattazione esatta, delegando la normalizzazione finale a script deterministici a valle, avvantaggia l'accuratezza complessiva dell'estrazione. Come illustrato nella figura 5.8, l'adozione dello schema tollerante comporta un miglioramento dell'F1-score medio per tutti i modelli testati.

Questo incremento, seppur modesto in termini assoluti, conferma che l'eliminazione dei vincoli sintattici rigidi riduce i fallimenti di validazione causati da lievi difformità formali, permettendo al modello di concentrare le proprie risorse computazionali sulla sola comprensione del testo.

Questo miglioramento diventa più netto analizzando il tasso di allucinazione (figura 5.9). Nello schema strict, quando il modello è incerto sul formato o sull'enum esatto richiesto (ad esempio, non «ricorda» la sigla esatta per la regione descritta nel testo), è probabilisticamente portato a «indovinare» un valore plausibile per soddisfare la richiesta sintattica, generando quella che, in uno schema rigido, viene classificata come un'allucinazione. Nello schema lenient, potendo semplicemente operare un copia-incolla semantico dal testo originale (es. restituendo «Emilia-Romagna»), il tasso di errore intrinseco si riduce sensibilmente: DeepSeek passa da 0,106 a 0,057 (-45,7%), e Llama scende da 0,142 a 0,107 (-24,1%).

Si evince quindi come, in una pipeline di estrazione strutturata, risulta più efficiente e affidabile delegare parte della normalizzazione a un componente deterministico,

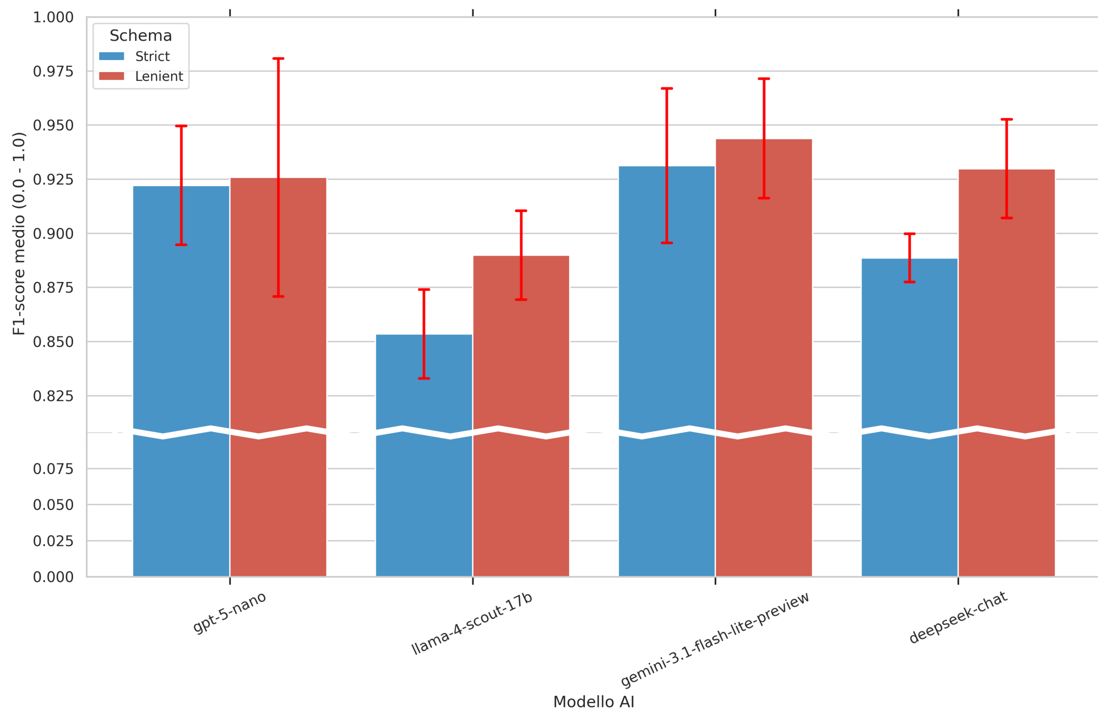

Figura 5.8: Confronto dell'F1-score medio per modello tra schema strict e lenient. Si nota un miglioramento generalizzato, particolarmente marcato nei modelli di dimensioni più contenute come DeepSeek-Chat e Llama-4-scout.

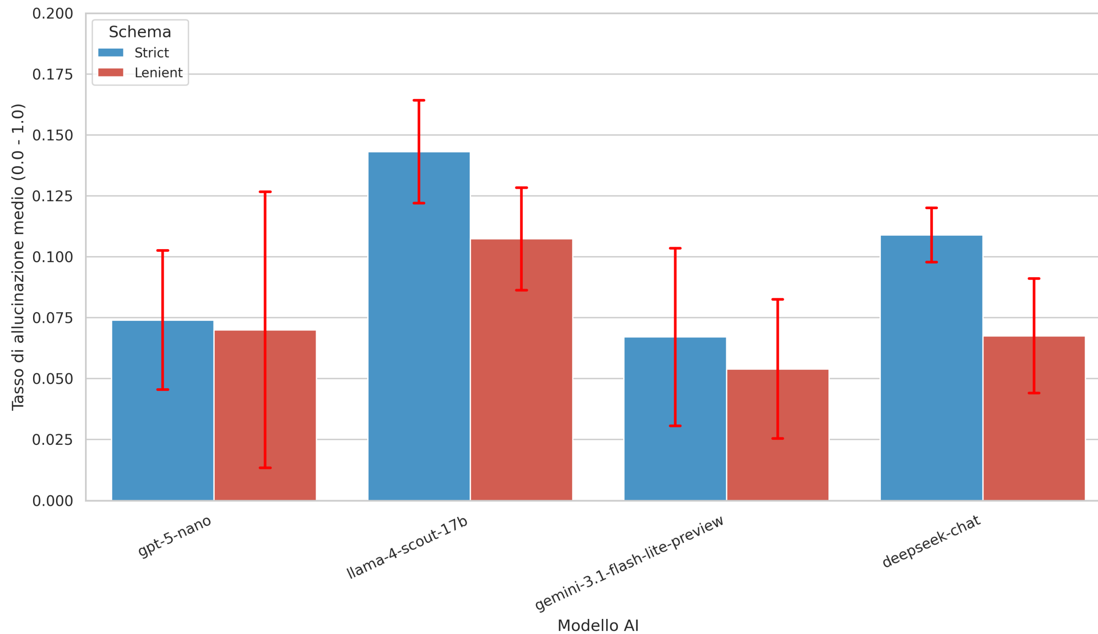

Figura 5.9: Tasso di allucinazione medio per modello (frequenza di inserimento di dati errati o non presenti nel testo). Lo schema lenient riduce significativamente gli errori forzati dal formato.

piuttosto che sovraccaricare il modello con requisiti sintattici che possono facilmente portare a errori evitabili.

5.6 Quantificare la resilienza strutturale

L'ultimo esperimento ha l'obiettivo di quantificare empiricamente la vulnerabilità strutturale dei parser tradizionali, ampiamente discussa nel capitolo 1 e nella sezione 2.1, confrontandola con la stabilità degli approcci basati sulla comprensione semantica.

Per quantificare questa vulnerabilità in modo controllato, il dataset originale è stato clonato e sottoposto alla funzione di *DOM chaos* descritta nella sezione 4.4.1. Questa procedura altera programmaticamente tutti i nomi delle classi CSS, rimpiazza i tag semantici (`<b>`, `<h4>`, `<ul>`) con tag contenitori generici e inserisce attributi casuali di disturbo, lasciando però del tutto inalterato il testo visibile a schermo.

I risultati del confronto, aggregati per metodo di estrazione, sono illustrati nella figura 5.10.

Il comportamento del parser manuale (implementato con espressioni regolari e libreria `Cheerio`) rappresenta la base di riferimento dell'esperimento. Sul DOM originale,

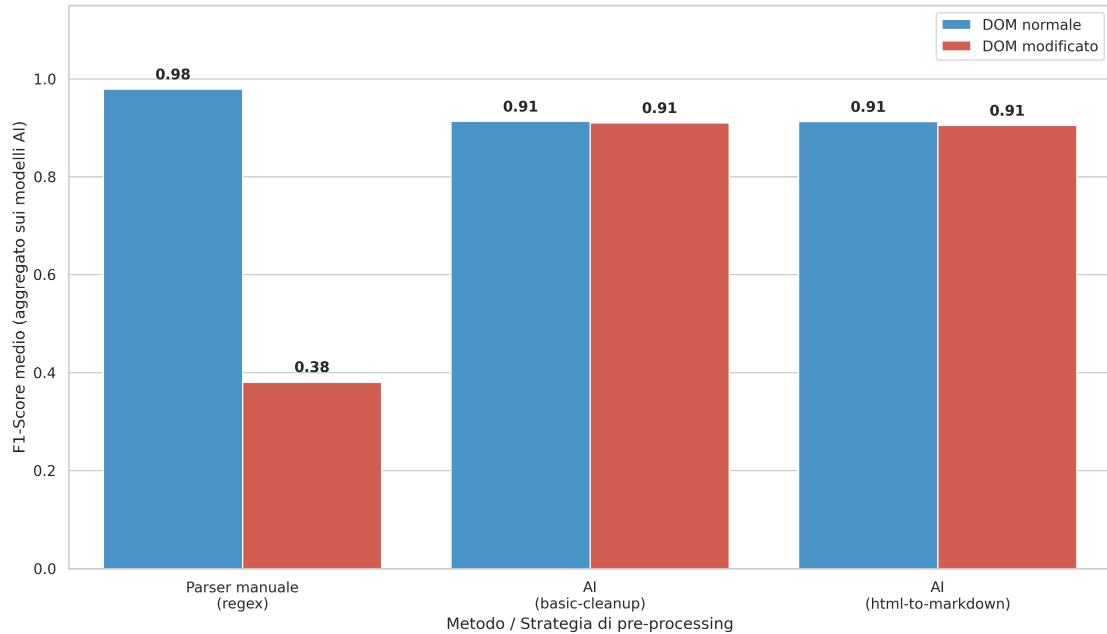

Figura 5.10: Confronto dell'F1-score medio su DOM intatto e DOM alterato (DOM chaos). Si osserva il forte degrado delle prestazioni del parser manuale, contrapposto alla stabilità assoluta degli approcci basati su intelligenza artificiale.

questo approccio ottiene un F1-score quasi perfetto (0,98), dimostrando l'estrema precisione raggiungibile quando la struttura del documento è nota. Tuttavia, sul dataset alterato, le performance subiscono un degrado netto, calando a un F1 di **0,38**.

Questo risultato conferma un limite intrinseco noto: le regole euristiche sono strettamente accoppiate alla struttura del documento. Il fatto che l'accuratezza non si sia azzerata del tutto è spiegabile dalla natura mista di questi parser: mentre i selettori CSS falliscono completamente a causa dell'offuscamento dei nomi delle classi (manifestando il tipico comportamento «fail-fast» menzionato nella sezione 2.4), alcune espressioni regolari progettate per operare su pattern di testo grezzo (ad esempio per catturare le date) riescono occasionalmente a «sopravvivere» e a catturare frammenti isolati. Ciononostante, una perdita di accuratezza superiore al 60% si traduce, in un ambiente di produzione, nella necessità di intervento manuale per correggere i selettori.

Al contrario, le strategie puramente automatiche basate su LLM si dimostrano totalmente immuni alla distruzione sintattica:

- La strategia `basic-cleanup` mantiene un F1-score identico (**0,91**) tra DOM

normale e modificato. Questo dimostra che il modello non fa alcun affidamento sulla gerarchia dei nodi o sulle classi, ma deduce le informazioni unicamente dalla semantica del contenuto testuale estratto.

- La conversione `html-to-markdown` mostra la medesima stabilità (restando a **0,91**), confermando che, anche in presenza di tag alterati (come alcuni header trasformati in `div` o in paragrafi), il modello riesce a interpretare correttamente la struttura logica del testo.

5.6.1 Osservazione sulla strategia ibrida

Un'importante osservazione emerge analizzando il comportamento della strategia ibrida `dom-distillation`. Di fronte al test del DOM chaos, la distillazione va inevitabilmente incontro allo stesso destino dello scraper manuale: non trovando più le classi CSS attese, il selettore fallisce. Nel framework di benchmark sviluppato per questa tesi, è stato implementato un meccanismo di fallback: qualora i selettori mirati non restituiscano alcun nodo, il sistema passa automaticamente alla strategia `basic-cleanup`. In un ambiente di produzione, questa scelta ingegneristica è auspicabile, poiché garantisce la continuità del servizio.

Dal punto di vista strettamente teorico, se non fosse stato implementato questo fallback, la funzione avrebbe estratto una stringa vuota, fornendo zero contesto all'LLM e registrando conseguentemente un F1-score pari a 0.

Questo test dimostra empiricamente il problema di partenza discusso nell'introduzione: qualsiasi componente dell'architettura di scraping che dipenda da vincoli strutturali del DOM (sia esso un parser interamente manuale o la fase preparatoria di una pipeline ibrida) costituisce un potenziale punto di rottura (*single point of failure*). Sebbene l'approccio totalmente automatico comporti oneri maggiori in termini di ottimizzazione dei token, si conferma l'unica via per ottenere un sistema di estrazione dati genuinamente «self-healing» e indipendente dalle evoluzioni grafiche del web.

Capitolo 6

Conclusioni e sviluppi futuri

Se per anni l'estrazione dei dati è stata una sfida di ingegneria puramente sintattica – volta a inseguire i cambiamenti strutturali del DOM tramite selettori sempre più complessi – l'avvento dei Large Language Model ha riorientato il paradigma più verso un'ingegneria di tipo semantico. I risultati empirici ottenuti tramite **SWEET** hanno permesso di quantificare non solo l'efficacia di questo cambio di paradigma, ma anche di definire le basi di una metodologia di estrazione più resiliente ed economica.

Alla luce delle evidenze sperimentali raccolte attraverso il framework di benchmark, è ora possibile rispondere puntualmente ai quesiti di ricerca formulati nell'introduzione di questo lavoro:

1. **Che impatto hanno le diverse strategie di preprocessing sul consumo di token e sull'accuratezza dei dati estratti?**
   L'analisi ha dimostrato che non esiste una strategia universalmente ottimale, ma un trade-off dipendente dal dominio. Inviare l'HTML grezzo è economicamente insostenibile e satura il contesto del modello. Alcune strategie di compressione estrema basate su SLM (come MinerU) riducono i token del 99%, ma causano un degrado dell'accuratezza (F1-score sotto lo 0,90) a causa della perdita di metadati contestuali vitali (es. tag temporali nascosti). All'estremo opposto, l'SLM Jina Reader (ReaderLM-v2) adotta un approccio conservativo che preserva quasi integralmente la struttura informativa originale, raggiungendo un F1-score superiore a 0,96 a fronte di una compressione più moderata. Le euristiche classiche, come la conversione in Markdown, offrono un ottimo compromesso per scenari generalisti. Tuttavia, per fonti semi-strutturate note, l'approccio ibrido della «DOM distillation» (pre-filtraggio tramite selettori CSS seguito da conversione in Markdown) si è rivelato la soluzione migliore: abbatte il consumo di token del 99,5% mantenendo l'F1-score costante-

mente superiore a 0,95, eliminando al contempo il rumore che può causare allucinazioni.

2. **In che misura la scelta del modello LLM e del formato di output influenza la qualità dei dati estratti?**
   L'indagine sperimentale ha evidenziato che la formulazione del prompt è il fattore più critico per il successo dell'estrazione. L'uso di istruzioni generiche ha mostrato limiti strutturali nella classificazione, mentre l'adozione di prompt espliciti con logiche di dominio ha permesso di abbattere drasticamente i falsi positivi riscontrati inizialmente. A parità di istruzioni, la dimensione del modello ha mostrato un impatto notevole sulle performance: architetture di scala intermedia (come Llama-4 Scout da 17B) hanno manifestato una propensione all'allucinazione superiore, con picchi del 14%, laddove modelli di scala maggiore (come DeepSeek-V3.2) hanno garantito un F1-score superiore a 0,95. Infine, la scelta dello schema di output (strict vs lenient) ha mostrato un impatto marginale sull'F1-score complessivo, ma si è rivelata determinante per la riduzione delle allucinazioni. L'approccio *lenient* permette infatti di recuperare estrazioni semantiche grezze e affidare la loro normalizzazione (date, codici ISTAT, province) a componenti deterministici a valle. Questo previene gli errori indotti dai vincoli di formato, che in modalità *strict* porterebbero il modello a «inventare» valori pur di soddisfare lo schema.

3. **È possibile quantificare il trade-off tra resilienza strutturale e costo computazionale/manutentivo?**
   La fragilità strutturale dei parser tradizionali è un limite intrinseco e ampiamente noto: qualsiasi alterazione del markup o dei selettori CSS invalida le regole di estrazione. Il test di alterazione sintattica (*DOM chaos*) è servito a quantificare empiricamente questa vulnerabilità, registrando il prevedibile crollo delle performance (F1-score da 0,98 a 0,38) a fronte di modifiche puramente strutturali invisibili all'utente. Al contrario, gli approcci basati su LLM, operando sulla comprensione semantica del contenuto testuale anziché sui percorsi sintattici, hanno dimostrato una totale immunità a tali variazioni (mantenendo un F1-score di 0,91). L'azzeramento di questo debito tecnico manutentivo impone tuttavia un trade-off operativo: l'adozione dell'IA sacrifica l'esecuzione deterministica, pressoché istantanea e a costo zero degli scraper tradizionali. L'inferenza semantica introduce infatti latenze significative (variabili da 1 a quasi 20 secondi per documento) e costi ricorrenti legati al consumo di token, rendendo questa soluzione eccellente per la stabilità nel tempo, ma meno adatta a scenari che richiedono estrazioni real-time o elaborazioni massive a budget limitato.

Come considerazione finale, sebbene i Large Language Model abbiano dimostrato ca-

pacità eccellenti nel parsing semantico e le tecniche di preprocessing basate su SLM siano destinate a evolversi ulteriormente, è doveroso ammettere che una pipeline di estrazione composta esclusivamente da intelligenza artificiale non rappresenta una soluzione universalmente ottimale. Come evidenziato dalla risposta al terzo quesito, ogni richiesta a un modello generativo comporta un onere architetturale: l'inferenza richiede infrastrutture computazionali significative (con conseguente consumo di risorse hardware ed energetiche nel caso di self-hosting, o costi diretti in caso di API commerciali) e impone latenze di svariati secondi che dipendono dalla dimensione del modello impiegato. Si tratta di dinamiche operative nettamente superiori rispetto all'estrazione tradizionale, che risulta pressoché istantanea e sostanzialmente gratuita a runtime.

Di conseguenza, le domande ingegneristiche fondamentali da porsi diventano: *quando ha realmente senso utilizzare gli LLM nel web scraping?* e *con quale livello di integrazione dovrebbero essere inseriti all'interno dell'architettura software?*

Torna utile, in questa sede, la tassonomia proposta da Flesca et al. [FMM+04], affrontata nella sezione 2.1. Come osservato, le pagine web possiedono gradi di strutturazione profondamente diversi. Per siti web appartenenti alla categoria delle pagine strutturate, o semi-strutturate con un layout stabile nel tempo, il parsing manuale rimane di gran lunga la tecnica più efficiente in termini economici, di latenza e di risorse computazionali. Di contro, applicare parser deterministici a pagine non strutturate – composte prevalentemente da testo libero, come i PDF di Trenitalia TPER analizzati in questo studio – rende il sistema incline all'errore e alla rottura, come dimostrato in modo emblematico dal caso di KDE Itinerary discusso nel capitolo 2.

L'intelligenza artificiale diventa vantaggiosa quando non si conosce a priori il layout della pagina (come nel caso dei web crawler) o quando l'informazione è espressa esclusivamente tramite linguaggio naturale. In questi scenari, l'integrazione dell'IA può assumere due forme. Un approccio *totale* prevede la completa sostituzione di selettori e regole fisse con l'uso esclusivo di modelli linguistici. Ciò richiede tecniche di pulizia automatizzate (euristiche o tramite SLM) per le quali, come emerso dai test, si è spesso costretti a sacrificare alcuni metadati periferici in cambio di un'elevata compressione, oppure di preservare l'integrità del contenuto a scapito di un maggior numero di token.

L'approccio alternativo è un uso *parziale* o ibrido: il software tradizionale seleziona le macro-aree contenenti il testo (come avviene nella *DOM distillation*), ma il parsing di dettaglio viene affidato all'LLM. Nel dominio specifico degli avvisi di sciopero del trasporto pubblico, quest'ultima emerge come l'architettura più efficace: le informazioni operative sono descritte in linguaggio naturale, il quale necessita di

comprensione semantica, ma i comunicati vengono pubblicati all'interno di portali istituzionali la cui struttura HTML cambia raramente nel tempo, giustificando un intervento misto che unisce l'affidabilità del codice tradizionale all'intelligenza adattiva dei modelli generativi.

Sulla base di queste considerazioni, si aprono diverse prospettive di sviluppo che potrebbero ulteriormente affinare l'architettura proposta. Una direzione promettente, parzialmente esplorata nelle fasi finali di questa ricerca, riguarda la scomposizione della pipeline in due stadi sequenziali: una prima fase di classificazione binaria veloce e una successiva di estrazione strutturata. La disponibilità di meccanismi di *context caching* (che consentono di riutilizzare i token in input tra chiamate successive con input simili), ormai supportati da molti provider di modelli linguistici, potrebbe rendere questo doppio passaggio tale da non incidere sensibilmente sul costo computazionale, migliorando al contempo la precisione nell'estrazione.

Inoltre, resta aperta la possibilità di applicare la capacità di «ragionamento» degli LLM per creare scraper «self-healing» in grado di adattarsi autonomamente ai cambiamenti strutturali delle pagine, come proposto da Kim et al. [KKJ25]. In questo paradigma, il sistema si affida ai selettori CSS della *DOM distillation* per il funzionamento ordinario; quando questi falliscono a seguito di una modifica del layout, l'LLM interviene per analizzare la nuova struttura e rigenerare automaticamente le regole di estrazione. L'intelligenza artificiale diventa così non tanto un motore di calcolo costante, quanto uno strumento di manutenzione automatizzata dei wrapper tradizionali.

Infine, l'estensione del metodo a domini meno strutturati degli avvisi di sciopero, come l'analisi di bandi di gara complessi o l'estrazione di informazioni da documenti legali, costituisce un interessante banco di prova per la generalizzazione di questo approccio. Con il maturare dei modelli multimodali, la frontiera dello scraping potrebbe spostarsi progressivamente dall'analisi del markup HTML verso la Computer Vision, interpretando lo screenshot della pagina esattamente come farebbe un utente umano, rendendo vane anche le più sofisticate tecniche di anti-scraping basate sull'offuscamento del codice.

Acronimi

AI Artificial Intelligence
API Application Programming Interface
CSS Cascading Style Sheets
CSV Comma-Separated Values
DOM Document Object Model
ETL Extract, Transform, Load
FN False Negative
FP False Positive
GPU Graphics Processing Unit
HTML HyperText Markup Language
IA Intelligenza Artificiale
IE Information Extraction
JSON JavaScript Object Notation
LLM Large Language Model
LPU Language Processing Unit
MD5 Message-Digest Algorithm 5
NLP Natural Language Processing
PDF Portable Document Format
regex Regular Expression
RPM Requests Per Minute
SDK Software Development Kit
SLM Small Language Model
SPA Single Page Application
SWEET Semantic Web Extraction & Evaluation Tool
TP True Positive
TPM Tokens Per Minute
TSP Tensor Streaming Processor

Riferimenti bibliografici

[ARS+20] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, and Brian Kurtz. *Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads*. May 2020. Pages: 158.

[AW24] Aman Ahluwalia and Suhrud Wani. Leveraging Large Language Models for Web Scraping, June 2024. arXiv:2406.08246 [cs].

[BDW26] Arth Bhardwaj, Nirav Diwan, and Gang Wang. Beyond BeautifulSoup: Benchmarking LLM-Powered Web Scraping for Everyday Users, January 2026. arXiv:2601.06301 [cs].

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners, July 2020. arXiv:2005.14165 [cs].

[BPK+25] Joe Barrow, Raj Patel, Misha Kharkovski, Ben Davies, and Ryan Schmitt. SafePassage: High-Fidelity Information Extraction with Black Box LLMs, September 2025. arXiv:2510.00276 [cs].

[DDL+24] John Dagdelen, Alexander Dunn, Sanghoon Lee, Nicholas Walker, Andrew S. Rosen, Gerbrand Ceder, Kristin A. Persson, and Anubhav Jain. Structured information extraction from scientific text with large language models. *Nature Communications*, 15(1):1418, February 2024.

[FMM+04] Sergio Flesca, Giuseppe Manco, Elio Masciari, Eugenio Rende, and Andrea Tagarelli. Web wrapper induction: A brief survey. *AI Commun.*, 17:57–61, January 2004.

[JLF+23] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Delong Chen, Wenliang Dai, Ho Shu Chan, Andrea Madotto, and Pascale Fung. Survey of Hallucination in Natural Language Generation. *ACM Computing Surveys*, 55(12):1–38, December 2023. arXiv:2202.03629 [cs].

[Kau24] Mavneet Kaur. HTML Web Scraping and Text Analysis: A Large Language Model Approach. Master's thesis, State University of New York at Stony Brook, United States – New York, 2024.

[KKJ25] Soyeon Kim, Namhee Kim, and Yeonwoo Jeong. NEXT-EVAL: Next Evaluation of Traditional and LLM Web Data Record Extraction, May 2025. arXiv:2505.17125 [cs].

[KS25] Navroz Kahlon and Williamjeet Singh. A Systematic Review of Web Scraping: Techniques, LLM-Enhanced Approaches, Performance Metrics, and Legal-Ethical Issues, September 2025.

[Kul26] Oleg Kulyk. From HTML to Embeddings - ML-Based Parsers That Survive Layout Changes | ScrapingAnt, January 2026.

[LPN+26] Mengjie Liu, Jiahui Peng, Wenchang Ning, Pei Chu, Jiantao Qiu, Ren Ma, He Zhu, Rui Min, Lindong Lu, Linfeng Hou, Kaiwen Liu, Yuan Qu, Zhenxiang Li, Chao Xu, Zhongying Tu, Wentao Zhang, and Conghui He. Dripper: Token-Efficient Main HTML Extraction with a Lightweight LM, March 2026. arXiv:2511.23119 [cs].

[MLC+23] Nick McKenna, Tianyi Li, Liang Cheng, Mohammad Javad Hosseini, Mark Johnson, and Mark Steedman. Sources of Hallucination by Large Language Models on Inference Tasks, October 2023. arXiv:2305.14552 [cs].

[noa19] Fix trenitalia pdf parsing (85b3080e) · Commits · PIM / KItinerary · GitLab, September 2019.

[WSW+25] Feng Wang, Zesheng Shi, Bo Wang, Nan Wang, and Han Xiao. ReaderLM-v2: Small Language Model for HTML to Markdown and JSON, March 2025. arXiv:2503.01151 [cs].

[YLZ+23] Hongbin Ye, Tong Liu, Aijia Zhang, Wei Hua, and Weiqiang Jia. Cognitive Mirage: A Review of Hallucinations in Large Language Models, September 2023. arXiv:2309.06794 [cs].

Appendici

Appendice A

Prompt di estrazione

In questa appendice vengono riportati testualmente i prompt di sistema utilizzati per istruire i Large Language Model durante l'esecuzione dei benchmark. Come discusso nel capitolo 5, l'evoluzione del prompt è risultata determinante per abbattere il tasso di *phantom strike* e migliorare l'accuratezza complessiva dell'estrazione.

I prompt sono stati implementati nel codice TypeScript come template string, in cui le variabili dinamiche (come `${sourceName}` o `${preProcessingStrategy}`) venivano interpolate a runtime dal motore di orchestrazione per fornire al modello il contesto specifico del documento analizzato.

A.1 Prompt generico (fase iniziale)

Questo prompt è stato utilizzato nella prima fase sperimentale. Fornisce istruzioni di base sul ruolo dell'agente e sui formati di output attesi, ma demanda interamente al modello la logica decisionale per la classificazione del documento. L'assenza di regole di dominio esplicite ha causato un tasso di allucinazioni intrinseche prossimo al 48% sui documenti ambigui (es. avvisi di revoca).

```
You are a precise data extraction algorithm.
Analyze the following content regarding a strike from "${sourceName}".

Extract the strike details according to the schema.
- Dates must be in 'yyyy-MM-dd HH:mm:ss'.
- Guaranteed times must be arrays of 'HH:mm-HH:mm' (omitted if not
↪  specified).
- Location codes should be: 2-digit ISTAT for regions (e.g., "03" for
↪  Lombardia), 2-letter for provinces (e.g., "MI" for Milan), or omitted
↪  for national strikes.
```

```
Input Content (Pre-processing: ${preProcessingStrategy}):
[... TESTO DEL DOCUMENTO ...]
```

A.2 Prompt esplicito (fase finale)

Sviluppato per mitigare le ambiguità semantiche, questo prompt incorpora le regole di dominio direttamente in linguaggio naturale. L'istruzione è strutturata in due fasi logiche: prima impone al modello di classificare la natura del documento escludendo esplicitamente revoche e report passati, e solo in caso positivo richiede l'estrazione. Inoltre, inietta il contesto geografico (`${companyContext}`) per prevenire allucinazioni estrinseche e fa uso della tecnica del *few-shot prompting*, fornendo esempi concreti di output JSON sia per i casi positivi che negativi.

```
You are a precise data extraction algorithm.
Analyze the following content regarding a possible strike from
↪  "${sourceName}" (${companyContext}).

IMPORTANT INSTRUCTIONS:
1. Decide if this document is actually announcing a new/upcoming strike.
  - If it is a cancellation/revocation of a strike, set isStrike to
   ↪  false.
  - If it is just providing information about participation (adesione) of
   ↪  a strike that already happened, set isStrike to false.
  - If it is providing real-time updates about an ongoing strike (e.g.,
   ↪  "the metro is currently closed due to a strike"), set isStrike to
   ↪  false.
  - If it is providing real-time service updates (e.g., metro is
   ↪  open/closed during the strike), set isStrike to false.
  - ONLY set isStrike to true if it is an announcement of an upcoming
   ↪  strike.

2. If isStrike is true, extract the strike details:
  - Dates must be in 'yyyy-MM-dd HH:mm:ss' format.
  - Guaranteed times must be arrays of 'HH:mm-HH:mm' (omitted if not
   ↪  specified).
  - Location codes should be: 2-digit ISTAT for regions (e.g., "03" for
   ↪  Lombardia), or omitted for national strikes.
```

```
Example Output if NOT a strike:
{
  "isStrike": false
}

Example Output if IS a strike:
{
  "isStrike": true,
  "strikeData": {
    "startDate": "2024-12-31 08:30:00",
    "endDate": "2024-12-31 12:30:00",
    "locationType": "REGIONAL",
    "locationCodes": ["12"],
    "guaranteedTimes": ["06:00-09:00", "18:00-21:00"]
  }
}

Input Content (Pre-processing: ${preProcessingStrategy}):
[... TESTO DEL DOCUMENTO ...]
```

Appendice B

Uso di strumenti di intelligenza artificiale generativa

Nel corso della redazione di questo lavoro, ho fatto uso di strumenti di intelligenza artificiale generativa, nello specifico Claude, DeepSeek e Gemini. Questi strumenti sono stati impiegati nel periodo settembre 2025 - marzo 2026, con l'obiettivo di supportare la scrittura del codice del framework di benchmark (in particolare l'implementazione dell'architettura e le espressioni regolari per i parser manuali), la ricerca di articoli scientifici e la revisione testuale di alcuni paragrafi per migliorarne la chiarezza espositiva. L'uso di tali modelli ha riguardato principalmente le sezioni relative allo «stato dell'arte», all'«implementazione» e alla «valutazione dei risultati».

Ringraziamenti

Un grazie a Giordano, compagno di avventure vissute durante l'Erasmus a Brno e di tutti i viaggi che abbiamo condiviso.

A mia madre che, nonostante le difficoltà, ha provato, continua, e so che continuerà a provare a camminare al mio fianco.

Un pensiero speciale va a Fiocchetto, che mi ha accompagnato per così tanto tempo. La tua assenza è un vuoto profondo in questo traguardo, ma il ricordo della tua presenza silenziosa resta un conforto prezioso. Ovunque tu sia, ti mando tutto il mio amore e tante carezze.