ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Dipartimento di Scienze
Corso di Laurea in Informatica

Progettazione e valutazione di interfacce dinamiche basate su LLM per task di troubleshooting fisico

Relatore:
Prof. Fabio Vitali

Presentata da:
Payam Salarieh

Correlatrice:
Dott.ssa Yu Mengyuan

I Sessione
Anno Accademico 2025/2026

"Make it simple, but significant."

Donald Draper

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Negli ultimi anni i Large Language Models e i modelli multimodali, come i Vision-Language Models, hanno assunto un ruolo sempre più importante nello sviluppo di sistemi interattivi. Questi modelli sono in grado di interpretare richieste in linguaggio naturale, generare testo e analizzare immagini.
La forma di interazione più comune solitamente risulta essere quella conversazionale, dove l'utente scrive una domanda o carica un contenuto, e il modello restituisce una risposta testuale. Tale approccio è semplice e flessibile, ma non è sempre il più adatto quando l'utente deve svolgere un'attività pratica composta da più passaggi.
In particolare, questo problema diventa evidente nei task di troubleshooting fisico, ovvero quelle attività in cui l'utente deve individuare la causa di un malfunzionamento ed eseguire una serie di controlli o azioni per provare a risolverlo. In questi scenari, invece di una semplice risposta testuale, può essere più utile una guida organizzata e interattiva.
L'impiego di LLM e VLM nei sistemi interattivi permette di andare oltre la semplice produzione di testo. I modelli possono interpretare un obiettivo, analizzare il contesto visivo e contribuire alla pianificazione di una sequenza di azioni, come avviene nei sistemi che interagiscono con interfacce grafiche (GUI) [Zha24; NCW24]. Inoltre, alcuni lavori hanno mostrato come l'integrazione dei modelli all'interno dell'interfaccia possa offrire forme di assistenza più contestuali e adattabili [OA24]. A partire da queste considerazioni, questa tesi propone *DynamicAI*, pensato per generare dinamicamente interfacce utente a supporto di task di troubleshooting fisico.

La difficoltà maggiore riguarda il supporto all'utente in attività fisiche di troubleshooting tramite strumenti statici o puramente conversazionali. Gli approcci più comuni

come manuali, guide online etc., possono essere più efficaci quando il problema è noto. Tuttavia, risultano meno adatti quando lo scenario è ambiguo, l'immagine è sfocata, l'utente non conosce la causa del malfunzionamento o quando sono necessari controlli progressivi prima di arrivare a una diagnosi accettabile.
Infatti, una guida statica propone solitamente lo stesso percorso a tutti gli utenti, lasciando all'utente il compito di cercare da solo la sezione più adatta al proprio caso, interpretare le istruzioni e capire quali passaggi siano effettivamente rilevanti. In un task fisico questa operazione può diventare difficile, soprattutto nel caso in cui l'utente non possieda le competenze tecniche; inoltre, una procedura statica risulterebbe inadattabile al feedback dell'utente: con il fallimento di un controllo o l'aggiunta di nuove informazioni, spesso risulta necessario tornare indietro, cercare un'altra guida o riformulare il problema.
Anche l'uso diretto di un chatbot basato su LLM presenta alcuni limiti: da un lato, il modello può generare spiegazioni utili e adattarsi alla richiesta dell'utente; dall'altro, la risposta prodotta rimane spesso una sequenza testuale. Questo può diventare meno efficace quando il task richiede conferme, scelte tra più alternative oppure controlli progressivi.
Per questo motivo, il problema affrontato in questa tesi non consiste quindi soltanto nella generazione di una procedura di troubleshooting, ma anche nella sua trasformazione in un'interfaccia strutturata e interattiva. Tale interfaccia deve poter rappresentare i passaggi proposti dal modello, raccogliere il feedback dell'utente e aggiornarsi quando vengono fornite nuove informazioni.

L'obiettivo principale di DynamicAI è quello di progettare e valutare un prototipo in grado di generare dinamicamente interfacce utente per task di troubleshooting fisico, utilizzando LLM/VLM per analizzare l'input iniziale e produrre una rappresentazione strutturata in formato JSON, che viene successivamente interpretata dal frontend e trasformata in una UI interattiva. In questo modo il modello viene utilizzato come supporto per costruire una procedura organizzata e modificabile.

A partire da questa idea, il lavoro si concentra su quattro aspetti principali:

- verificare se un sistema basato su LLM/VLM, sia effettivamente in grado di generare una struttura UI dinamica valida per i task di troubleshooting fisico
- valutare la funzionalità della UI generata e la sua efficacia nel guidare l'utente in diversi tipi di scenari di troubleshooting fisico

- analizzare il miglioramento della qualità e dell'utilità dell'interfaccia generata, quando viene applicato un processo di refinement basato sul feedback dell'utente
- individuare i principali limiti dell'approccio proposto, in termini di: latenza, affidabilità e dipendenza da modelli o provider esterni

L'interazione proposta da DynamicAI, non si basa su un chatbot tradizionale, ma su un'interfaccia dinamica composta da sezioni, controlli e messaggi contestuali, che sostituiscono il concetto di produrre una semplice risposta testuale. Il sistema, dunque, genera una UI interattiva che permette all'utente di seguire una procedura, selezionare opzioni e fornire nuove informazioni durante il troubleshooting del task.
Per rappresentare tale interfaccia, viene utilizzata una struttura JSON contenente una descrizione astratta della UI, al posto di produrre direttamente codice frontend, che viene successivamente interpretata da una libreria controllata di componenti; questa scelta separa la generazione logica dell'UI dal suo rendering grafico, mantenendo più controllabile il comportamento del sistema.
Dal punto di vista architetturale, il sistema realizzato è organizzato in modo tale da separare il frontend, il backend e il provider del modello LLM/VLM. Il frontend si occupa della visualizzazione dell'interfaccia e della raccolta del feedback dell'utente, mentre il backend gestisce la sessione, la costruzione del prompt, la comunicazione con il modello e il controllo della risposta.
Dopo la prima generazione della UI, l'utente ha la possibilità di interagire con i componenti prodotti e di fornire un feedback, mirando ad aggiornare la versione dell'interfaccia. In questo modo, il sistema mantiene una *history* della sessione del processo di troubleshooting. La UI non viene quindi considerata come un risultato definitivo, ma come una struttura modificabile durante il processo di risoluzione del task. Di conseguenza, questo processo si collega agli approcci basati su workflow adattivi e a controlli contestuali, nei quali l'interazione non viene definita una volta sola, ma può essere aggiornata progressivamente in base alle informazioni raccolte durante l'esecuzione [LXM24; DWS25].
Infine, la valutazione del sistema è stata condotta su scenari rappresentativi di troubleshooting fisico, includendo sia problemi direttamente visibili sia situazioni più ambigue, che richiedono una procedura esplorativa. In questo modo è stato possibile confrontare la generazione iniziale con le successive fasi di refinement e osservare anche i principali limiti pratici emersi durante l'interazione con il modello.
Nei capitoli successivi della tesi, vengono presentati e approfonditi diversi aspetti del lavoro svolto.

Il **Capitolo 2** introduce i principali riferimenti delle ricerche teoriche e progettuali utili a collocare il lavoro, concentrandosi sul tema delle interfacce dinamiche basate su LLM e VLM. In particolare, vengono discussi il ruolo dei modelli linguistici e multimodali nei sistemi interattivi, la capacità di interpretare input testuali e visivi, e il modo in cui questi modelli possono essere utilizzati per adattare meglio l'interazione rispetto alle interfacce tradizionali.
L'analisi dei lavori collegati ai GUI agents, alla generazione di workflow e ai controlli contestuali, sposta l'attenzione su come tali approcci abbiano contribuito alla produzione di testo verso interazioni più strutturate e modificabili. Questa analisi permette di collocare DynamicAI all'interno di un ambito più ampio, nel quale l'interfaccia non è definita una volta sola, ma può essere generata e aggiornata in base al contesto e al feedback dell'utente.

Il **Capitolo 3** descrive l'architettura di DynamicAI e le principali scelte progettuali adottate durante lo sviluppo del prototipo. Dalla visione del flusso del sistema, vengono analizzati i passaggi che portano dall'input iniziale dell'utente alla generazione dell'interfaccia: l'immagine o la descrizione del problema viene inviata al backend, interpretata dal modello LLM/VLM e trasformata in una rappresentazione JSON della UI. Tale rappresentazione viene poi elaborata dal frontend e convertita in componenti grafici interattivi. Il capitolo approfondisce inoltre la scelta di utilizzare una struttura JSON invece della generazione diretta di codice frontend, l'uso di una libreria controllata di componenti, la gestione dello stato della sessione e il meccanismo di fallback previsto nei casi in cui l'output del modello non sia utilizzabile. Viene infine descritto il processo di refinement, attraverso il quale l'interfaccia può essere aggiornata sulla base del feedback fornito dall'utente.

Il **Capitolo 4** presenta la valutazione di DynamicAI, con l'obiettivo di verificare che vengano date risposte, in misura soddisfacente, agli aspetti individuati nell'introduzione. Inizialmente vengono definite le metriche utilizzate per analizzare il comportamento del sistema, prendendo in considerazione la correttezza strutturale dell'output, la qualità funzionale delle interfacce, il contributo del refinement, e alcuni aspetti pratici legati alle prestazioni e all'affidabilità dell'integrazione con provider esterni. Successivamente viene descritta la metodologia adottata, basata su scenari rappresentativi di troubleshooting fisico, nel quale vengono confrontati sia problemi direttamente riconoscibili dall'immagine, sia situazioni più ambigue. Inoltre, i risultati ottenuti permettono di confrontare la generazione iniziale con le versioni successive delle interfacce aggiornate tramite feed-

back, osservando in quali casi DynamicAI riesce a produrre una guida utile e in quali situazioni emergono limiti o comportamenti non adeguati.

Infine, il **Capitolo 5** riassume i risultati principali del lavoro e discute il contributo del prototipo rispetto agli obiettivi iniziali. Vengono ripresi gli aspetti emersi durante la progettazione e la valutazione, con particolare attenzione alla capacità del sistema di generare interfacce strutturate, alla funzione del refinement nel migliorare progressivamente la UI e ai limiti pratici osservati durante i test. La parte finale si conclude indicando alcune possibili direzioni future, tra cui l'uso di modelli più avanzati, l'estensione della valutazione a scenari più complessi e il miglioramento della gestione dello stato dell'interfaccia durante il processo di troubleshooting.

Capitolo 2

Interfacce dinamiche basate su LLM e VLM

In questo capitolo vengono introdotti i concetti chiave alla base del progetto, come i modelli LLM/VLM, la generazione dinamica di interfacce e il troubleshooting guidato.

2.1 LLM e VLM nei sistemi interattivi

I Large Language Models (LLMs) sono modelli di Artificial Intelligence (AI) utilizzati per elaborare e generare grandi quantità di testo, con l'obiettivo di comprendere e imparare il linguaggio umano. A partire da un'istruzione testuale, possono produrre risposte, riassunti, spiegazioni o sequenze di operazioni coerenti con il contesto fornito. Il loro funzionamento si basa sull'elaborazione di rappresentazioni testuali, suddivise in token, che vengono utilizzate dal modello per individuare relazioni tra le parole e generare l'output successivo.
I Vision-Language Models (VLM) estendono queste capacità integrando anche informazioni visive. Oltre al testo, il modello può quindi ricevere immagini, schermate o fotografie e mettere in relazione ciò che viene osservato con la richiesta dell'utente. Questa combinazione permette di affrontare compiti nei quali una parte rilevante delle informazioni non è espressa direttamente in forma testuale.

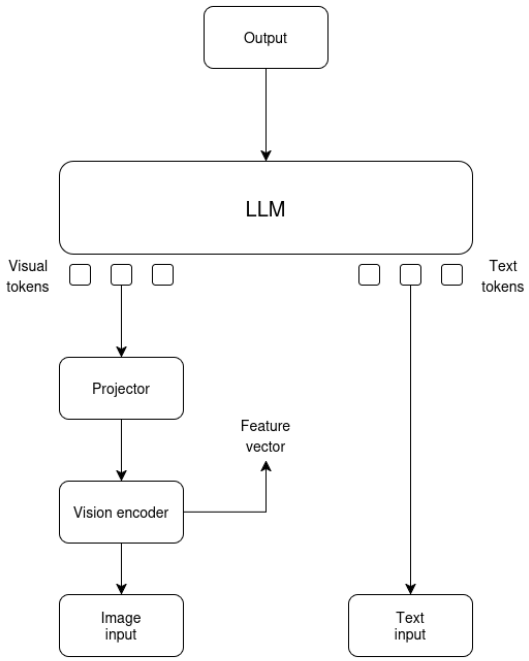


Figura 2.1: Schema semplificato dell'elaborazione congiunta [IBM26]

2.1.1 Interpretazione e gestione dell'interazione

Nei sistemi interattivi basati su LLM/VLM, l'utente può esprimere un obiettivo in linguaggio naturale senza dover specificare in anticipo ogni singola operazione necessaria, e successivamente il modello interpreta la richiesta, individua le informazioni rilevanti e trasforma l'obiettivo generale in una sequenza di passaggi più semplici. Un esempio significativo è rappresentato dai *GUI agents*, sistemi progettati per interagire con applicazioni web, desktop o mobili. In questo caso, il modello riceve un'istruzione dell'utente e osserva lo stato dell'interfaccia. Sulla base di queste informazioni deve riconoscere gli elementi disponibili, comprenderne la funzione e scegliere quali azioni eseguire per completare il compito [Zha24]. Il processo non consiste soltanto nel produrre un piano iniziale. L'interfaccia può cambiare dopo ogni azione, mostrando nuovi contenuti, messaggi di errore o opzioni che prima non erano disponibili. Per questo motivo il sistema deve osservare nuovamente lo stato corrente e decidere se proseguire con il piano originale oppure modificarlo.
Il funzionamento di questi agenti può essere descritto attraverso quattro capacità principali, collegate tra loro [NCW24]:

- **Percezione**: acquisisce le informazioni presenti nell'interfaccia
- **Ragionamento**: mette in relazione le informazioni osservate con l'obiettivo espresso dall'utente
- **Pianificazione**: organizza le operazioni necessarie per raggiungere l'obiettivo, stabilendone l'ordine
- **Azione**: traduce il piano in operazioni concrete sull'interfaccia, come la selezione di un elemento o l'inserimento di un valore

Queste capacità formano un processo iterativo. Dopo l'esecuzione di un'azione, il nuovo stato dell'interfaccia viene nuovamente osservato e può determinare la continuazione o la modifica del piano iniziale.

2.1.2 Opportunità e limiti

La combinazione tra comprensione linguistica e analisi visiva permette di affrontare attività nelle quali le informazioni necessarie sono distribuite tra la richiesta dell'utente e ciò che viene mostrato in un'immagine o in un'interfaccia. Il modello può quindi utilizzare il contesto osservato per individuare gli elementi rilevanti e produrre indicazioni più adatte alla situazione corrente [Zha24; NCW24].
L'affidabilità del risultato dipende tuttavia dalla correttezza della percezione e del ragionamento. Il riconoscimento errato di un elemento può influenzare la pianificazione e portare all'esecuzione di azioni non appropriate. Questo problema diventa più rilevante nei compiti composti da molti passaggi, nei quali un errore iniziale può propagarsi nelle fasi successive.

Ulteriori difficoltà riguardano la varietà delle interfacce, la gestione di situazioni non previste e la possibilità che il modello produca indicazioni plausibili ma non corrette. Per questo motivo, nei sistemi interattivi è necessario prevedere meccanismi di controllo dell'output e di aggiornamento dello stato durante l'esecuzione.

2.2 Generazione e adattamento delle interfacce

Le interfacce tradizionali sono generalmente definite in fase di sviluppo: struttura, componenti e possibilità di interazione vengono stabiliti in anticipo e rimangono sostanzialmente invariati durante l'utilizzo. Questo approccio è adeguato quando le attività

e le esigenze dell'utente sono prevedibili, ma risulta meno flessibile quando il contenuto dell'interazione dipende dal contesto o dalle informazioni fornite durante l'esecuzione. L'integrazione degli LLM nelle interfacce permette di introdurre forme di assistenza contestuale. Anziché limitarsi a mostrare una struttura fissa, il sistema può utilizzare le informazioni disponibili per fornire indicazioni o controlli collegati all'attività corrente. Un approccio di questo tipo è stato sperimentato integrando il supporto del modello direttamente all'interno di interfacce esistenti, con l'obiettivo di guidare l'utente senza sostituire completamente l'applicazione originale [OA24].

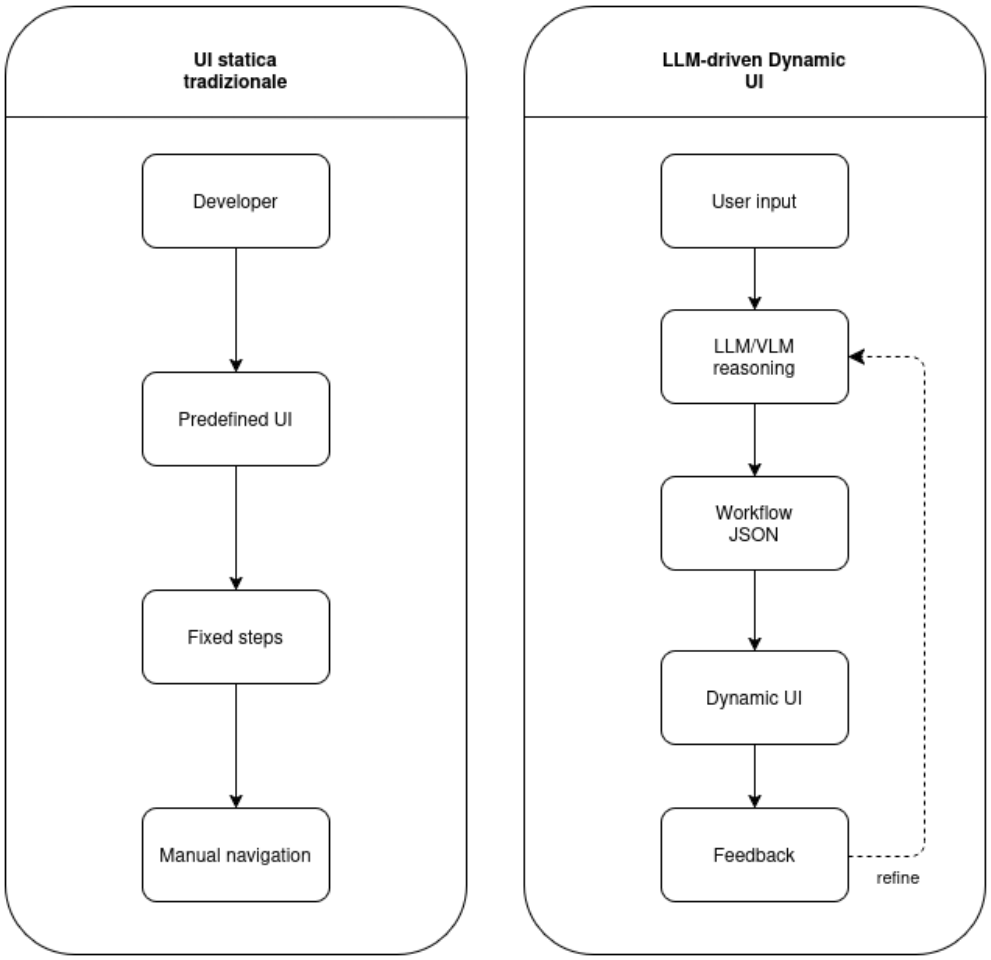


Figura 2.2: Confronto concettuale tra un flusso di interazione statico e un flusso adattato al contesto.

La Figura 2.2 mostra come in un'interfaccia statica il flusso viene definito interamente

prima dell'esecuzione, mentre in un approccio dinamico, il contenuto dell'interazione può dipendere dall'input iniziale, dallo stato corrente e dalle informazioni raccolte nei passaggi successivi.

2.2.1 Workflow e adattamento dell'interazione

Quando un'attività richiede più operazioni, il comportamento del sistema può essere organizzato attraverso un *workflow*, cioè una sequenza di passaggi collegati tra loro e orientati al raggiungimento di un obiettivo. Nei sistemi basati su LLM, tale sequenza non deve necessariamente essere definita interamente in anticipo, ma può essere costruita o modificata sulla base dell'input ricevuto e dei risultati prodotti durante l'esecuzione.
La generazione automatica di workflow permette di scomporre un compito complesso in operazioni più semplici e di stabilire le dipendenze tra i diversi passaggi. Alcuni approcci prevedono inoltre una fase di ottimizzazione iterativa, nella quale il workflow iniziale viene valutato e successivamente modificato per migliorarne l'efficacia rispetto al compito assegnato [LXM24].
Questo tipo di organizzazione è utile soprattutto quando non è possibile conoscere fin dall'inizio tutte le operazioni necessarie. L'esito di un passaggio può fornire nuove informazioni, rendere superflua un'azione prevista oppure richiedere ulteriori verifiche. Il workflow assume quindi una struttura adattabile, nella quale le decisioni successive dipendono dallo stato raggiunto.

La rappresentazione grafica del processo può rendere più comprensibili le relazioni tra i diversi passaggi e consentire all'utente di intervenire direttamente sulla loro composizione. Le interazioni visuali permettono, ad esempio, di aggiungere, modificare o collegare operazioni basate su LLM senza descrivere l'intero processo esclusivamente attraverso un prompt testuale [CMW24].
In questo modo, il workflow non rimane una sequenza interna al sistema, ma diventa una parte visibile dell'interazione. L'utente può comprenderne meglio la struttura, controllare i singoli passaggi e correggere più facilmente eventuali risultati non adeguati.
L'adattamento può riguardare anche il modo in cui la richiesta viene raffinata durante l'interazione. Quando il risultato iniziale non è sufficientemente preciso, l'uso esclusivo del testo richiede spesso di riscrivere o ampliare il prompt, con il rischio di ripetere informazioni già fornite o modificare involontariamente parti corrette della richiesta.

L'inserimento di controlli contestuali nell'interfaccia permette invece di precisare soltanto gli aspetti che richiedono una modifica. Il feedback dell'utente viene così trasfor-

mato in un aggiornamento mirato del prompt e può essere utilizzato nelle elaborazioni successive [DWS25].

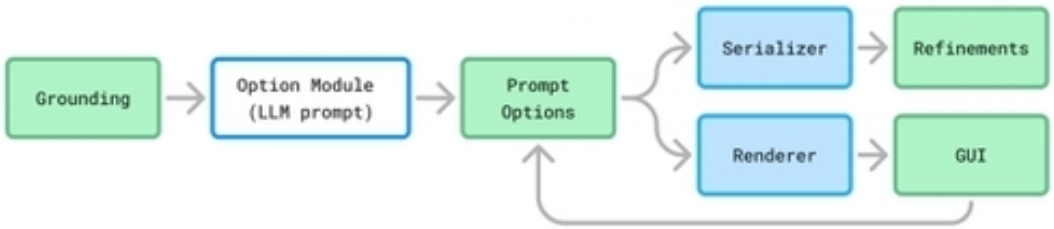


Figura 2.3: Architettura del processo di generazione e raffinamento dei controlli contestuali [DWS25].

La Figura 2.3 mostra che il contesto iniziale viene utilizzato per generare un insieme di opzioni di raffinamento. Tali opzioni vengono sia serializzate per aggiornare il prompt, sia renderizzate sotto forma di componenti grafici. Le scelte effettuate dall'utente attraverso la GUI vengono quindi reinserite nel processo, creando un ciclo iterativo di aggiornamento dell'interazione. Il processo non produce quindi necessariamente un risultato definitivo in un singolo passaggio. Lo stato corrente, gli esiti intermedi e le indicazioni fornite dall'utente possono contribuire progressivamente alla definizione del workflow e delle operazioni successive.

2.3 Troubleshooting guidato

Il troubleshooting può essere considerato un processo decisionale nel quale si cerca di individuare la causa di un malfunzionamento a partire da informazioni incomplete. Il problema osservato non conduce sempre a una sola spiegazione: sintomi simili possono dipendere da cause differenti e, prima di intervenire, è spesso necessario verificare più ipotesi.
Una procedura guidata organizza questo processo attraverso una successione di osservazioni, controlli e possibili interventi. Ogni verifica produce nuove informazioni, che possono confermare un'ipotesi, escluderla oppure indicare la necessità di ulteriori controlli. Il percorso di risoluzione dipende quindi non soltanto dal problema iniziale, ma anche dagli esiti ottenuti durante l'esecuzione.

2.3.1 Caratteristiche dei task fisici

Nei task fisici, lo stato del sistema non è interamente disponibile in forma digitale. La diagnosi può dipendere da elementi visibili, dalla disposizione dei componenti, dalla qualità dei collegamenti o dal comportamento osservato durante una prova. Una descrizione testuale può risultare incompleta, mentre una fotografia mostra soltanto una parte della situazione e non consente sempre di verificare condizioni interne o dinamiche.

L'utente svolge quindi un ruolo attivo nel processo diagnostico, dove può essere necessario chiedergli di controllare un collegamento, osservare una variazione, eseguire una prova o fornire una nuova immagine. Le informazioni raccolte durante queste operazioni modificano progressivamente la rappresentazione del problema e permettono di restringere le possibili cause.
Le istruzioni devono inoltre essere formulate in modo operativo, dato che non è sufficiente indicare una possibile causa; occorre specificare quale elemento osservare, quale azione compiere e come interpretarne il risultato. La chiarezza della procedura assume particolare importanza quando l'utente non possiede competenze tecniche e deve operare direttamente su un oggetto reale.

2.3.2 Limiti degli approcci tradizionali

Manuali e guide di assistenza descrivono generalmente problemi già classificati e associano a ciascuno una sequenza di istruzioni. Questa organizzazione rende semplice la consultazione quando il sintomo è riconoscibile e la procedura corrisponde esattamente allo scenario dell'utente.
Le difficoltà emergono quando il problema non rientra chiaramente in una categoria. L'utente deve interpretare i sintomi, confrontarli con le descrizioni disponibili e scegliere autonomamente il percorso più appropriato. Se la selezione iniziale è errata, può seguire una procedura non pertinente senza ottenere informazioni utili alla diagnosi.
Le procedure lineari presentano inoltre una capacità limitata di reagire agli esiti intermedi. Un passaggio può rivelare che una determinata causa è da escludere, ma la guida continua comunque secondo il flusso definito in precedenza. Allo stesso modo, può essere necessario eseguire un controllo non previsto o tornare a una fase precedente.

Il limite principale non risiede quindi nella correttezza delle singole istruzioni, ma nella difficoltà di selezionarle e organizzarle in base allo stato specifico del problema.

Nei contesti incerti, il supporto deve poter utilizzare le informazioni raccolte durante l'esecuzione per determinare quali verifiche proporre successivamente.

Capitolo 3

DynamicAI: architettura del sistema

A partire dai limiti delle procedure statiche e dalla capacità inferiore delle guide tradizionali di adattarsi alle informazioni raccolte durante l'interazione, è stato sviluppato un sistema denominato ***DynamicAI***. Tale prototipo integra un modello LLM/VLM per generare e aggiornare interfacce grafiche in funzione del problema osservato e del feedback dell'utente.

3.1 Requisiti e scelte progettuali

La progettazione di DynamicAI è stata pianificata tenendo conto di un insieme di requisiti funzionali e non funzionali, definiti a partire dall'obiettivo principale del sistema, ovvero quello di supportare l'utente nello svolgimento di task di troubleshooting fisico attraverso un'interfaccia dinamica non basata necessariamente su una conversazione testuale.
Tra i requisiti principali rientrano la validità dell'output prodotto dal modello, la possibilità di mantenere lo stato della sessione, la gestione degli errori e la separazione tra interfaccia, logica applicativa e accesso al modello. Inoltre, il sistema deve evitare che un output non valido produca una schermata vuota, ricorrendo a messaggi di errore o a un'interfaccia generica di fallback.
Tra le scelte progettuali più rilevanti adottate, vi è l'utilizzo di una rappresentazione di un oggetto JSON dell'interfaccia, anziché nella generazione diretta di codice HTML o React [Met26]. Il frontend interpreta questa struttura, la trasforma in componenti grafici e associa ogni elemento a un componente di una libreria chiusa, comprendente gruppi

di selezione, pulsanti, slider, menu, aree di testo, avvisi e schede informative. Frontend e backend sono stati sviluppati in modo indipendente, mentre la chiave API utilizzata per accedere a OpenRouter [Ope26] viene gestita tramite variabili d'ambiente in un file `.env` e caricata esclusivamente dal backend, evitando di inserirla direttamente nel codice sorgente per non esporla a minacce esterne.

3.2 Struttura del sistema

DynamicAI è organizzato in tre componenti principali: frontend, backend e modello LLM/VLM accessibile tramite OpenRouter. Il frontend gestisce l'interazione con l'utente, il caricamento dell'input, la visualizzazione dell'interfaccia generata e la raccolta del feedback. Il backend rappresenta il livello intermedio tra frontend e modello: crea e mantiene la sessione, costruisce il prompt, invia la richiesta al provider, controlla la risposta ricevuta e restituisce al frontend una struttura JSON utilizzabile. Il modello LLM/VLM, infine, interpreta l'input iniziale o il feedback dell'utente e produce la descrizione dell'interfaccia da renderizzare.

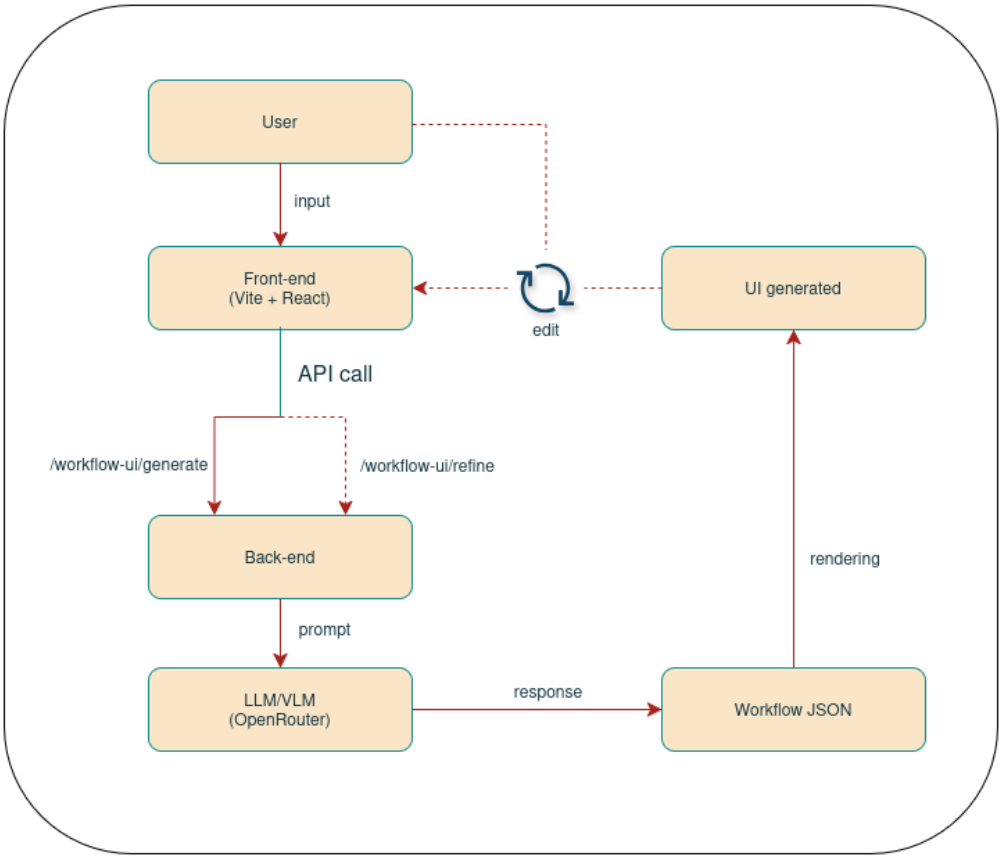


Figura 3.1: Architettura generale del sistema di generazione dinamica di interfacce utente basata su LLM/VLM

La Figura 3.1 mostra il flusso principale del sistema. L'utente fornisce un input al frontend, che invia una richiesta al backend attraverso gli endpoint dedicati alla generazione iniziale e al refinement. Il backend costruisce il prompt, comunica con il modello tramite OpenRouter e riceve una risposta strutturata. Tale risposta viene trasformata in un Workflow JSON, che il frontend utilizza per renderizzare l'interfaccia dinamica.

3.2.1 Frontend

Vite [Vit26] e React sono gli strumenti principali utilizzati per lo sviluppo dell'interfaccia utente. Vite è stato scelto per la velocità di sviluppo e facilità di configurazione,

mentre React per la sua architettura basata su componenti, che si adatta alla costruzione di interfacce dinamiche e modulari.

Il compito principale del frontend, dunque, è trasformare la struttura JSON ricevuta dal backend in componenti grafici e visualizzabili dall'utente. Ogni componente descritto nel JSON viene associato a un componente React corrispondente. In questo modo, il frontend non contiene una procedura fissa, ma costruisce l'interfaccia in base all'output generato dal modello.

Configurazione del server di sviluppo

Durante lo sviluppo locale, il frontend è stato configurato tramite il file `vite.config.js`. Questo file definisce l'utilizzo del plugin React e permette di personalizzare il comportamento del server di sviluppo di Vite.

In particolare, la configurazione può essere utilizzata per gestire la comunicazione tra frontend e backend, ad esempio inoltrando le richieste API verso il server FastAPI. In questo modo, il frontend può chiamare gli endpoint del backend senza dover gestire direttamente problemi legati al CORS durante la fase di sviluppo.

```
1   server: {
2     proxy: {
3       "/api": {
4         target: "http://127.0.0.1:8000",
5         changeOrigin: true,
6       },
7     },
8   },
```

Listing 3.1: Configurazione Vite del frontend

3.2.2 Backend

La comunicazione tra l'interfaccia utente e il modello linguistico non viene gestita direttamente dal browser, ma passa attraverso un layer applicativo intermedio. Questa scelta permette di proteggere la chiave API, centralizzare la costruzione del prompt e controllare la validità delle risposte ricevute.

Il servizio backend, realizzato in Python [Pyt26] con FastAPI [Ram26], espone gli endpoint necessari per la generazione iniziale e per il refinement dell'interfaccia. Una volta ricevuta la richiesta dal frontend, il backend costruisce il prompt, invia la chiamata a OpenRouter e restituisce al frontend una struttura JSON normalizzata.

3.2.3 Integrazione del modello linguistico

La scelta del modello linguistico giusto è fondamentale, poiché in base alla sua capacità di interpretare il task fornito dall'utente, riesce a produrre una rappresentazione strutturata della UI. Per evitare che il frontend dipenda direttamente dal modello, la comunicazione viene gestita dal backend, che costruisce il prompt, invia la richiesta al modello e restituisce al frontend un output JSON normalizzato.

Durante lo sviluppo, sono stati considerati due approcci principali:

- **Test in locale tramite Ollama** [Oll26]: in una prima fase sono stati sperimentati modelli eseguiti localmente, tra cui *Qwen3-VL* e *Qwen3.5:2B*. Questo approccio avrebbe permesso di ridurre la dipendenza da servizi esterni e di mantenere maggiore controllo sull'esecuzione del modello. Tuttavia, i test locali hanno evidenziato alcuni limiti pratici: i modelli multimodali risultavano più lenti e più pesanti dal punto di vista delle risorse richieste, mentre i modelli più leggeri erano più rapidi ma meno adatti a produrre output strutturati complessi.
- **Chiamate API tramite OpenRouter**: nella versione finale è stata privilegiata l'integrazione tramite OpenRouter, che permette di accedere a diversi modelli linguistici attraverso un'unica API. Questa soluzione semplifica la sperimentazione con modelli differenti e consente di modificare il modello utilizzato agendo principalmente sulla configurazione del backend. Il backend invia a OpenRouter il prompt, le informazioni relative al task e i vincoli sul formato dell'output, richiedendo una risposta in formato JSON utilizzabile dal frontend.

La scelta finale di utilizzare OpenRouter è stata quindi motivata dalla maggiore flessibilità dell'integrazione e dalla possibilità di sperimentare più facilmente modelli diversi senza modificare la struttura complessiva del sistema.

3.3 Libreria dei componenti

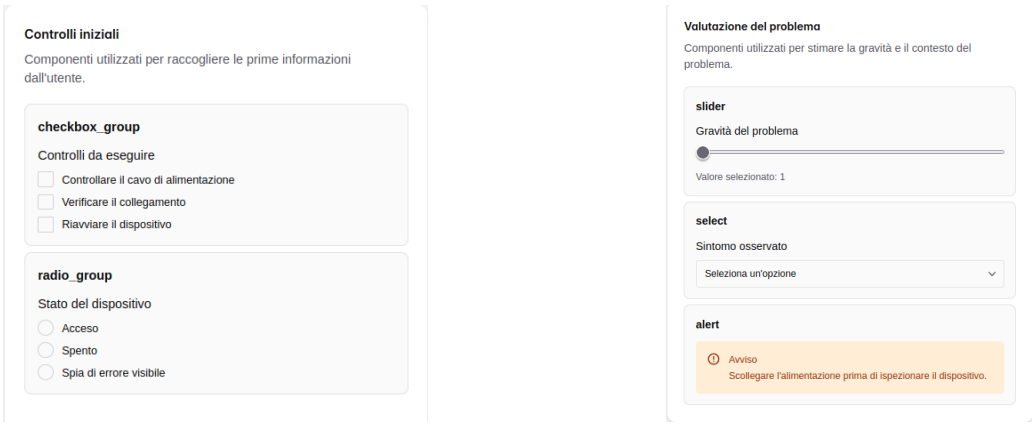
La libreria dei componenti rappresenta l'insieme degli elementi grafici che il sistema può utilizzare per costruire l'interfaccia dinamica. Nel prototipo sviluppato, i componenti sono implementati nel frontend React utilizzando Chakra UI, una libreria che fornisce componenti già pronti e personalizzabili.

Dunque, il modello non genera direttamente codice React o HTML, ma produce una descrizione astratta dell'interfaccia in formato JSON. Ogni componente indicato nel JSON viene poi associato dal frontend a un componente React corrispondente. In questo modo, il modello può scegliere quali elementi usare e quali contenuti mostrare, mentre il frontend mantiene il controllo sulla resa grafica e sul comportamento dell'interfaccia.

I componenti implementati nel prototipo includono:

- `checkbox_group`: permette all'utente di selezionare più opzioni contemporaneamente. È utile per rappresentare controlli multipli da eseguire, ad esempio verificare cavi, alimentazione o stato dei componenti.
- `radio_group`: permette di scegliere una sola opzione tra più alternative. È adatto quando le risposte sono mutuamente esclusive, ad esempio indicare se un dispositivo è acceso, spento o in errore.
- `button_group`: rappresenta un insieme di azioni rapide o scelte operative. Può essere usato per far selezionare all'utente una direzione della procedura, ad esempio riprovare un controllo o passare alla fase successiva.
- `slider`: consente di indicare un valore su una scala graduata. È utile per rappresentare intensità, livello di confidenza o gravità percepita del problema.
- `select`: permette di scegliere un'opzione da un elenco. È indicato quando le alternative disponibili sono numerose e occuperebbero troppo spazio se mostrate tutte direttamente.
- `textarea`: consente all'utente di inserire testo libero, ad esempio dettagli aggiuntivi sul problema o feedback dopo un controllo.
- `alert`: mostra messaggi importanti, avvisi o suggerimenti. Può essere usato per evidenziare un possibile rischio, una condizione critica o un'informazione da non ignorare.

- `info_card`: presenta informazioni descrittive o istruzioni contestuali. È utile per spiegare brevemente una fase della procedura o fornire indicazioni generali.



Controlli iniziali Valutazione del problema

Figura 3.2: Esempi di componenti della libreria dinamica renderizzati dal frontend.

3.4 Prompt e formato dell'output

Il prompt ha il compito di guidare il modello nella generazione di un'interfaccia strutturata. Invece di richiedere una risposta testuale libera, il backend specifica chiaramente il formato dell'output, i componenti disponibili e le regole che il modello deve rispettare.

Questa impostazione permette di ottenere risposte più controllabili e più facilmente utilizzabili dal frontend. Il prompt include le istruzioni generali sul ruolo del modello, la descrizione del task fornito dall'utente, l'elenco dei componenti ammessi e la struttura JSON attesa.

3.4.1 Struttura JSON

L'interfaccia generata è rappresentata tramite un oggetto JSON. La struttura contiene un titolo generale, una descrizione e un insieme di sezioni. Ogni sezione rappresenta una parte della procedura e contiene uno o più componenti grafici.

Un esempio semplificato della struttura è il seguente:

```
{
  "title": "Diagnosi del problema",
```

```
  "description": "Interfaccia guidata per il troubleshooting.",
  "sections": [
    {
      "id": "section-1",
      "title": "Controlli iniziali",
      "description": "Verificare le condizioni di base.",
      "components": [
        {
          "component": "checkbox_group",
          "label": "Controlli da eseguire",
          "options": ["Verificare il cavo", "Controllare l'alimentazione"]
        }
      ]
    }
  ]
}
```

3.4.2 Generazione iniziale

A partire dall'input fornito dall'utente, il sistema costruisce una prima rappresentazione completa dell'interfaccia. In questa fase il modello deve interpretare il problema, individuare una possibile sequenza di controlli e scegliere i componenti più adatti per rappresentare ogni parte della procedura.
Si tratta della fase più aperta del processo, perché l'intera struttura della UI viene prodotta a partire da poche informazioni iniziali. Per ridurre risposte ambigue o non utilizzabili, il prompt impone vincoli espliciti sul formato JSON e sui tipi di componenti ammessi.

Il prompt della generazione iniziale definisce innanzitutto il ruolo del modello, specificando che deve agire come un sistema per la progettazione di interfacce dinamiche orientate a task fisici di troubleshooting. Successivamente vengono indicati i vincoli principali dell'output: la risposta deve essere un oggetto JSON valido, composto da un titolo, un sommario e un insieme di sezioni. Ogni sezione contiene una descrizione e una lista di componenti scelti esclusivamente dalla libreria supportata dal frontend.
Un estratto semplificato del prompt utilizzato nella generazione iniziale è il seguente:

```
Sei un sistema LLM per la progettazione di interfacce dinamiche
```

```
per task fisici di troubleshooting.

L'utente ha caricato uno o più file.
Il tuo obiettivo non è produrre una risposta da chatbot
e non è generare un flusso conversazionale passo dopo passo.
Devi invece inferire un possibile workflow di task
e trasformarlo in una singola interfaccia dinamica
composta da più sezioni strutturate.

Genera un'interfaccia orientata al task con:

-un titolo
-un breve sommario
-da 2 a 4 sezioni
-componenti UI scelti dalla libreria consentita

Tipi di componenti consentiti:
checkbox_group, radio_group, button_group, slider,
select, textarea, alert, info_card.

Le sezioni devono riflettere una sequenza logica o funzionale
del troubleshooting.
I componenti devono aiutare l'utente a controllare,
scegliere, confermare o fornire nuove informazioni.

Restituisci solo JSON valido.
Non usare markdown.
Non restituire messaggi da chat.
Mantieni l'interfaccia coerente, compatta e in italiano.
```

Listing 3.2: Estratto del prompt di generazione iniziale

Il prompt viene impostato per limitare la libertà del modello e rendere la risposta più controllata, in maniera da ragionare internamente sul problema e sulla possibile sequenza di controlli. Il risultato finale deve essere una UI strutturata e non una spiegazione testuale libera, per permettere al backend di verificare la risposta ricevuta e al frontend di trasformarla in componenti grafici. Inoltre, include anche alcune regole di qualità. In particolare, il modello viene istruito a basare l'interfaccia sugli elementi visibili o inferibili dall'immagine caricata, evitando procedure troppo generiche. Le sezioni devono avere una funzione concreta nel troubleshooting e i componenti devono essere scelti in modo coerente con il tipo di decisione richiesta: ad esempio, una checklist può essere usata

per controlli multipli, un radio group per scelte mutuamente esclusive e uno slider per valutazioni graduali.

3.4.3 Refinement dell'interfaccia

Dopo la generazione iniziale, l'utente può fornire nuove informazioni o richiedere modifiche alla UI prodotta. In questa fase il modello riceve sia il feedback sia la struttura corrente dell'interfaccia, e deve produrre una versione aggiornata coerente con lo stato precedente [DWS25].
Rispetto alla generazione iniziale, il refinement è un compito più vincolato: il modello non deve ricostruire l'interfaccia da zero, ma intervenire su una struttura già esistente. Questo consente modifiche più mirate e riduce il rischio di perdere informazioni utili già presenti nella UI.
Un estratto semplificato del prompt di refinement è il seguente:

```
Hai già generato una prima interfaccia strutturata.
Ora devi aggiornarla sulla base del feedback dell'utente.

Interfaccia attuale:
{current_ui}

Feedback raccolto finora:
{feedback_state}

Obiettivo:

generare una versione aggiornata dell'interfaccia
migliorare l'efficacia della UI per guidare l'utente
usare il feedback per restringere le ipotesi
aggiornare le sezioni successive o incomplete
mantenere coerenza con l'interfaccia precedente
mostrare un avanzamento reale verso la risoluzione
non trasformare l'interfaccia in una chat
non rigenerare arbitrariamente tutto da zero

Se il feedback permette di chiarire meglio il problema,
rendi i controlli più specifici.
Se alcune verifiche non sono più necessarie,
rimuovile o sostituiscile con passaggi più mirati.

Restituisci solo JSON valido con la struttura prevista.
```

```
Mantieni tutto in italiano.
Evita componenti ridondanti o troppo generici.
```

Listing 3.3: Estratto del prompt di refinement

Questo processo permette a DynamicAI di trattare l'interfaccia come una struttura modificabile. Fornendo un'informazione aggiuntiva, il modello può aggiornare le sezioni della UI, rimuovere controlli non più necessari o aggiungere verifiche più specifiche. Quindi, la funzione del refinement non diventa solo quella di correggere eventuali errori, ma di far avanzare la procedura di troubleshooting in base ai feedback raccolti.
Dal punto di vista del sistema, anche l'output del refinement deve rispettare lo stesso formato JSON utilizzato nella generazione iniziale. Questo consente al frontend di riutilizzare lo stesso meccanismo di *rendering dinamico*, indipendentemente dal fatto che l'interfaccia sia stata prodotta per la prima volta o aggiornata dopo il feedback dell'utente. In questo modo, il processo rimane uniforme: il backend costruisce il prompt, il modello genera una nuova descrizione della UI e il frontend la visualizza attraverso i componenti della libreria controllata.

3.5 Esecuzione multi-step

Per illustrare completamente il funzionamento di DynamicAI, si può considerare un flusso di interazione multi-step, relativo a uno scenario di troubleshooting fisico. L'esempio non ha lo scopo di valutare formalmente il sistema, ma di mostrare come le diverse parti dell'architettura collaborano durante l'interazione: caricamento dell'input, interpretazione da parte del modello, generazione della UI, raccolta del feedback e aggiornamento dell'interfaccia. In base allo scenario, l'utente carica un'immagine relativa a un problema fisico, e passando dal frontend al backend, viene creata una nuova sessione, dove viene costruito il prompt da fornire al modello LLM/VLM. Il modello analizza il contenuto visivo e produce una descrizione strutturata dell'interfaccia in formato JSON, contenente le sezioni della procedura, i componenti da mostrare e i testi associati ai controlli. Il frontend interpreta questa struttura, associa ogni componente al corrispondente elemento della libreria React e renderizza una UI interattiva. Dunque, la prima interfaccia può includere una diagnosi iniziale, una checklist di verifiche, alcune opzioni per indicare il punto in cui si manifesta il problema e un messaggio di avviso relativo alle precauzioni da seguire.
Dopo aver eseguito i primi controlli, l'utente può fornire nuove informazioni attraverso

i componenti della UI, in modo da inviare nuovi dati al backend insieme allo stato corrente dell'interfaccia. I dati inviati vengono utilizzati nel processo di refinement, dove il modello non riparte da zero, ma per merito della history della sessione, riceve la UI già generata e il feedback dell'utente, producendo una versione aggiornata dell'oggetto JSON. In questo modo, DynamicAI tratta la UI come una struttura modificabile durante il processo di troubleshooting, mantenendo continuità tra la generazione iniziale e gli aggiornamenti successivi.

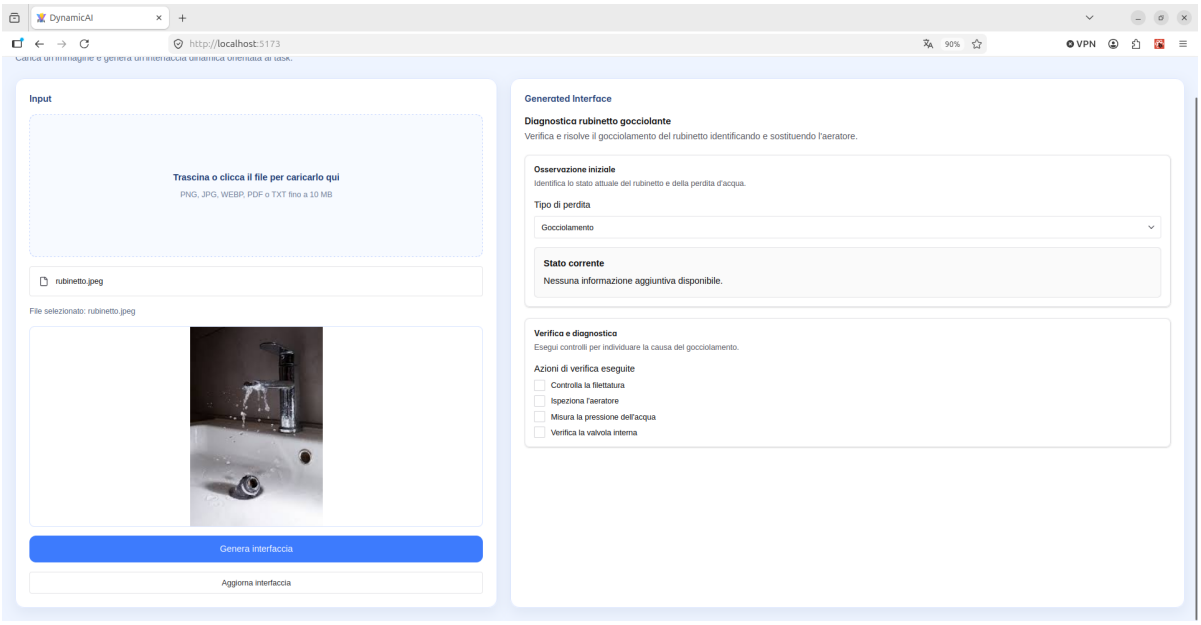


Figura 3.3: Prima interfaccia generata da DynamicAI a partire dall'immagine del rubinetto.

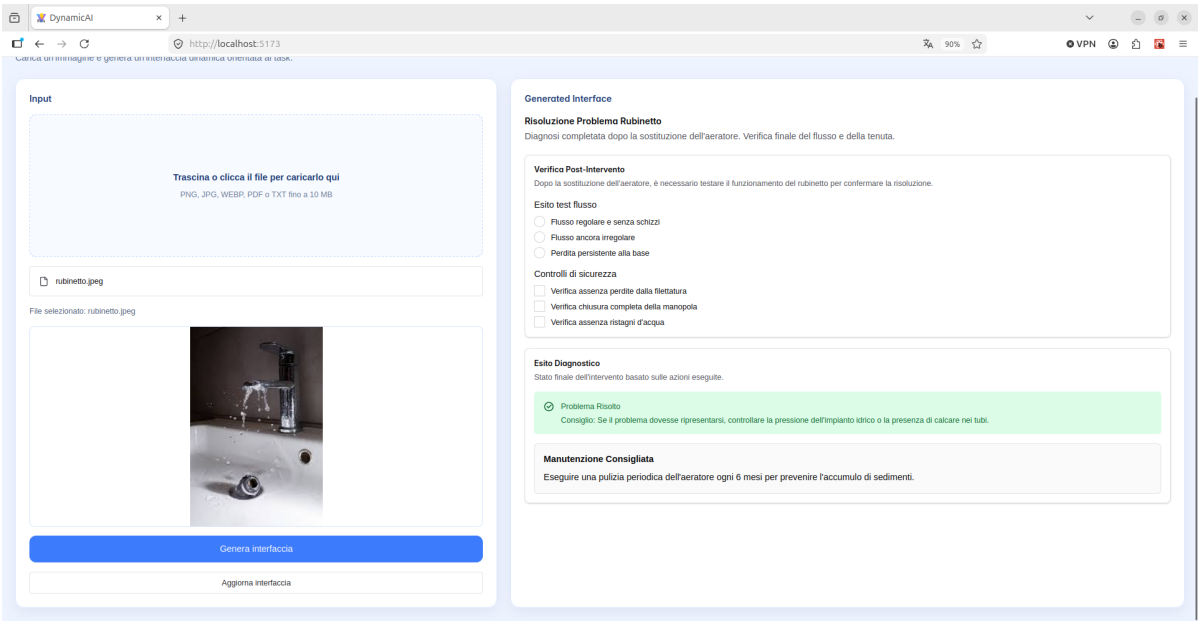


Figura 3.4: Interfaccia aggiornata dopo il feedback dell'utente tramite processo di refinement.

3.6 Evoluzione del prototipo

Durante lo sviluppo del progetto, è stata adottata una metodologia iterativa, che ha permesso di evolvere il prototipo in modo progressivo, integrando feedback continui e migliorando le funzionalità in base alle esigenze emerse durante la fase di test. Sono state considerate due diverse impostazioni del modello, dove la prima versione del prototipo si focalizzava su una guida step-by-step, nella quale l'utente riceveva progressivamente un passaggio alla volta. La versione finale, invece, si basa su una UI dinamica multi-step, nella quale l'utente riceve un'interfaccia con una struttura più completa, che include più passaggi e opzioni per esplorare diverse soluzioni al problema presentato.

3.6.1 Primo prototipo

Il primo prototipo era orientato alla generazione progressiva di istruzioni, dove l'utente forniva un input e il sistema rispondeva con un singolo passaggio o con una porzione limitata della procedura di troubleshooting.

Questa impostazione era utile per sperimentare la decomposizione del task, ma presentava alcuni limiti.

In particolare, la guida risultava fortemente sequenziale e dipendente da interazioni successive. Inoltre, l'interfaccia generata non rappresentava ancora una vera UI dinamica, ma piuttosto una forma strutturata di risposta guidata.

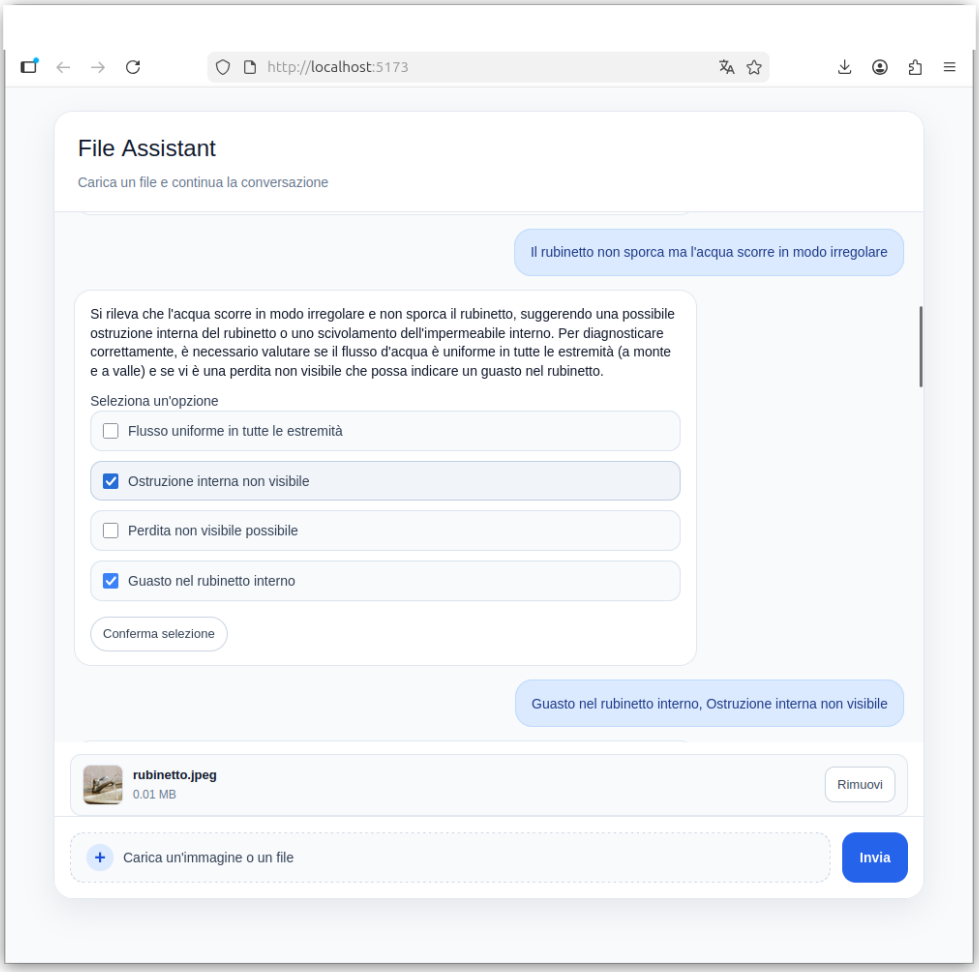


Figura 3.5: Schermata del primo prototipo basato su guida step-by-step.

3.6.2 Versione finale

La versione finale di DynamicAI supera questi limiti, proponendo una UI dinamica multi-step che fornisce all'utente un'interfaccia più completa e flessibile fin dalla prima generazione.

In questo caso, il modello riceve il task e produce un oggetto JSON che descrive sezioni, componenti e contenuti della UI. Successivamente il frontend interpreta questa struttura e la trasforma in una GUI interattiva, composta da elementi come bottoni, checklist, aree di testo, alert e slider.
Questo approccio permette di rappresentare il task in modo più ricco rispetto a una semplice risposta testuale. Inoltre, il processo di refinement consente di aggiornare l'interfaccia sulla base del feedback dell'utente, mantenendo una struttura coerente con la procedura già generata. Nel complesso, la versione finale rappresenta quindi il passaggio da una guida sequenziale a un'interfaccia dinamica e modificabile, più coerente con l'obiettivo di supportare task di troubleshooting fisico attraverso componenti interattivi generati in base al contesto.

Capitolo 4

Test e valutazione

Per analizzare al meglio l'efficacia del sistema proposto, è stata condotta una serie di test su scenari rappresentativi di troubleshooting fisico.

La valutazione considera sia la generazione iniziale dell'interfaccia sia il processo di refinement, attraverso il quale una UI già generata viene modificata in base al *feedback* da parte dell'utente. Questa distinzione permette di analizzare non solo la qualità della prima risposta del sistema, ma anche la sua capacità di migliorare progressivamente l'interfaccia prodotta.

4.1 Obiettivi di valutazione

L'analisi sperimentale è stata organizzata con l'obiettivo di verificare il comportamento del prototipo in scenari rappresentativi di troubleshooting fisico. In particolare, sono stati considerati quattro aspetti principali:

- **Correttezza strutturale dell'output**: verificare se il sistema produce una UI compatibile con il formato JSON atteso dal frontend e composta da componenti appartenenti alla libreria implementata.
- **Qualità funzionale dell'interfaccia**: valutare se i controlli, le istruzioni e le azioni proposte siano coerenti con lo scenario di troubleshooting e sufficienti per guidare l'utente verso una possibile diagnosi o soluzione.

- **Efficacia del refinement**: confrontare la generazione iniziale con le interfacce aggiornate tramite feedback dell'utente, osservando se il processo di refinement consente di correggere, completare o rendere più specifica la UI prodotta.
- **Prestazioni e affidabilità**: analizzare i tempi di risposta delle chiamate API e gli eventuali errori restituiti dal provider esterno, poiché questi aspetti influenzano l'utilizzabilità del sistema in un contesto interattivo.

Questi obiettivi permettono di distinguere tra problemi tecnici dell'output, legati alla validità della struttura generata, e problemi funzionali, legati invece alla pertinenza della procedura proposta rispetto allo scenario analizzato.

4.2 Metriche di valutazione

Per valutare il lavoro svolto dal sistema e lo sviluppo del prototipo, sono state definite delle metriche sia tecniche che funzionali. Vengono quindi considerati alcuni aspetti legati alla correttezza strutturale dell'interfaccia generata, alla qualità funzionale in termini di aderenza alle esigenze dell'utente e alla capacità di guidare l'utente verso una possibile soluzione, e infine alle prestazioni del sistema in termini di tempo di risposta.

4.2.1 Correttezza strutturale

Dal punto di vista tecnico, una UI generata può essere utilizzata solo se rispetta il formato atteso dal frontend. Il sistema, infatti, non interpreta testo libero, ma una struttura JSON composta da sezioni e componenti predefiniti.
In questa categoria sono stati considerati due aspetti principali: la validità della struttura prodotta e la compatibilità dei componenti generati con la libreria implementata. Un'interfaccia è stata considerata valida quando conteneva le informazioni necessarie alla visualizzazione, rispettava l'organizzazione prevista in sezioni e componenti, e non includeva tipi di elementi non supportati dal frontend.

La valutazione è stata espressa in forma binaria: valore 1 per una UI strutturalmente valida, valore 0 per una UI non valida o non renderizzabile correttamente. Questa metrica permette di distinguere gli errori tecnici dell'output dagli errori legati al contenuto della procedura.

4.2.2 Qualità funzionale

Una volta verificata la validità tecnica dell'interfaccia, è necessario valutare se il contenuto generato sia effettivamente utile rispetto allo scenario di partenza. Questa parte della valutazione riguarda quindi la capacità del sistema di proporre controlli, istruzioni e passaggi coerenti con il problema descritto.

La qualità funzionale è stata valutata manualmente, confrontando la UI generata con il comportamento atteso per ciascuno scenario. In particolare, sono stati considerati tre aspetti: la pertinenza dei passaggi proposti, l'ordine logico della procedura e la capacità dell'interfaccia di guidare l'utente verso una possibile diagnosi o soluzione.
Per ridurre la soggettività della valutazione, ogni output è stato analizzato secondo criteri ricorrenti. Un passaggio è stato considerato corretto quando rappresentava un controllo o un'azione coerente con il problema. Al contrario, sono stati considerati problematici i passaggi troppo generici, non collegati allo scenario o potenzialmente fuorvianti.
L'accuratezza della decomposizione del task è stata stimata come rapporto tra il numero di passaggi corretti individuati nella UI e il numero di passaggi attesi per lo scenario considerato:

$$Accuracy = \frac{Passaggi\ corretti}{Passaggi\ attesi}$$

Il task success rate indica invece se l'interfaccia, nel suo complesso, è stata ritenuta sufficiente per supportare l'utente nella risoluzione del problema. Questa metrica è stata espressa in forma binaria: valore 1 quando la UI era complessivamente adeguata allo scenario, valore 0 quando risultava insufficiente o non coerente.

Poiché queste metriche dipendono da una valutazione manuale, i risultati devono essere interpretati come indicatori qualitativi e non come misure assolute. Per questo motivo, nella discussione dei risultati viene dato maggiore peso al confronto tra generazione iniziale e refinement e all'analisi dei casi rappresentativi, piuttosto che al singolo valore numerico.

4.2.3 Prestazioni

Oltre alla qualità dell'interfaccia prodotta, è stato misurato anche il tempo necessario per ottenere una risposta dal modello linguistico. Questa misura è rilevante perché il

sistema è pensato per un utilizzo interattivo: latenze troppo elevate possono rendere meno fluida l'esperienza utente, soprattutto durante il refinement.

La latenza è stata calcolata come il tempo trascorso tra l'invio della richiesta al modello e la ricezione della risposta completa da parte del backend. Per ogni chiamata API è stato registrato il tempo di risposta in secondi.
A partire dai log raccolti sono stati calcolati media, mediana, valore minimo e valore massimo. La media fornisce un'indicazione generale del tempo di risposta, mentre la mediana è utile per interpretare i risultati in presenza di outlier. I valori minimo e massimo permettono invece di osservare la variabilità delle chiamate e individuare casi anomali.

Durante l'analisi sono stati registrati anche gli stati delle chiamate API, distinguendo tra risposte completate correttamente e chiamate terminate con errore. Gli errori del provider non sono stati considerati come fallimenti qualitativi della UI, ma come elementi rilevanti per valutare l'affidabilità complessiva dell'integrazione con servizi esterni.

4.3 Metodologia

La metodologia di valutazione è stata definita con l'obiettivo di osservare il comportamento del sistema in scenari rappresentativi di troubleshooting fisico. Ogni scenario descrive un problema concreto che richiede una sequenza di controlli o azioni da svolgere passo dopo passo.

Per ciascuno scenario è stata richiesta al sistema la generazione di un'interfaccia dinamica multi-step. L'interfaccia prodotta è stata successivamente analizzata in base alle metriche definite nella sezione seguente. In alcuni casi, a partire dalla UI generata, sono state eseguite ulteriori chiamate di refinement per valutare la capacità del sistema di modificare e migliorare l'interfaccia sulla base di nuove istruzioni.

4.3.1 Scenari e procedura di test

Gli scenari di test sono stati scelti per valutare le capacità del sistema in diversi contesti di troubleshooting fisico.
Le prove non presentavano tutte lo stesso tipo di problema, come nel caso della stampante e del setup incompleto, dove il problema non era visibile, perciò il sistema si adattava

a una procedura più esplorativa, basata su domande e risposte per identificare la causa del problema.

Al contrario, negli scenari del rubinetto, del motore, del cavo scollegato e del router, il problema era visibile, quindi il sistema si focalizzava su una procedura più diretta, con istruzioni specifiche per risolvere il problema identificato.

4.3.2 Numero di test effettuati

Complessivamente sono stati analizzati 94 test relativi alla qualità delle interfacce generate. Di questi, 32 test riguardano la generazione iniziale e 62 test riguardano il refinement dell'interfaccia.

La distribuzione dei test non è uniforme tra generazione iniziale e refinement, poiché una singola interfaccia generata può essere raffinata più volte. In alcuni scenari, infatti, il processo di refinement è stato ripetuto per simulare una progressiva acquisizione di nuove informazioni sul problema o per correggere aspetti incompleti della UI iniziale.

La distribuzione dei test per scenario è riportata nella Tabella 4.1.

ID	**Scenario**	**Generazione iniziale**	**Refinement**	**Totale**
S1	Stampante	7	9	16
S2	Setup incompleto	3	6	9
S3	Rubinetto	6	10	16
S4	Motore	5	13	18
S5	Cavo scollegato	6	14	20
S6	Router	5	10	15
Totale	–	**32**	**62**	**94**

Tabella 4.1: Distribuzione dei test effettuati per scenario e fase di valutazione.

Il numero maggiore di test di refinement riflette la natura iterativa del sistema. Dopo una prima generazione, l'interfaccia può essere modificata più volte sulla base di feedback successivi. Per questo motivo, il refinement non è stato valutato come un singolo passaggio isolato, ma come un processo di miglioramento progressivo della UI generata.

Modelli utilizzati

Modello	**Gemma 3 4B**	**Gemma 3 12B**	**Gemma 3 27B**	**Gemma 4 26B**
Tipo	LLM	LLM	LLM	LLM
Validità UI	A volte incoerente	Valida	Corretta	Valida
Correttezza linguistica	Discreta	Coerente	Valida	Corretta
Struttura JSON generata	Spesso corretta	Valida	Efficace e coerente	Aderente al formato richiesto
Esperienza utente	Incompleta	Valida	Corretta	Valida

Tabella 4.2: Valutazione qualitativa dei modelli Google utilizzati tramite OpenRouter.

Modello	**Nemotron Nano 12B VL**	**Nemotron 3 Nano Omni**	**Qianfan OCR Fast**
Tipo	VLM	Multimodale / reasoning	OCR
Validità UI	Valido, ma poco stabile	Media	Non adatta
Correttezza linguistica	Troppo generico a volte	Buona	Errata
Struttura JSON generata	Formato non sempre rispettato	Variabile, a volte incompleta	Scarsa e inadeguata
Esperienza utente	Media	Media	Bassa

Tabella 4.3: Valutazione qualitativa dei modelli Nvidia e Baidu utilizzati tramite OpenRouter.

Dai test effettuati, i modelli Google sono risultati complessivamente più precisi nella generazione di interfacce coerenti e strutturate. I modelli Nvidia hanno mostrato risultati intermedi, con output talvolta utili ma meno stabili. Il modello Baidu, invece, si è dimostrato poco adatto alla generazione completa della UI, risultando più limitato rispetto agli altri modelli considerati.

4.4 Risultati

Qualità funzionale

La qualità funzionale è stata valutata confrontando le interfacce generate con il comportamento atteso per ciascuno scenario.

Fase	**Accuratezza media stimata**
Generazione iniziale	$0.691 \approx 69.1\%$
Refinement	$0.903 \approx 90.3\%$

Tabella 4.4: Accuratezza media nelle fasi di generazione iniziale e refinement.

Fase	**Task success rate stimato**
Generazione iniziale	$0.656 \approx 65.6\%$
Refinement	$0.935 \approx 93.5\%$

Tabella 4.5: Task success rate nelle fasi di generazione iniziale e refinement.

I risultati riportati nelle Tabelle 4.4 e 4.5 mostrano un miglioramento evidente nella fase di refinement rispetto alla generazione iniziale. Questi risultati indicano che la prima generazione dell'interfaccia produce spesso una base utile e coerente, ma non sempre completa o sufficientemente dettagliata, infatti, in alcuni casi l'interfaccia iniziale non considera pienamente alcuni dettagli del problema.
Nonostante ciò, la procedura di refinement ottiene risultati migliori, poiché opera su una UI già esistente e riceve informazioni aggiuntive dall'utente, permettendo di correggere sezioni già prodotte oppure completare parti mancanti.
Il miglioramento è particolarmente rilevante negli scenari in cui il problema non è immediatamente evidente, come nel caso della stampante o del setup incompleto. In questi casi, il feedback dell'utente permette al sistema di restringere progressivamente le ipotesi e proporre controlli più mirati. Negli scenari con problemi più visibili, invece, la generazione iniziale risulta spesso già abbastanza efficace, ma il refinement può comunque migliorare la precisione delle istruzioni.

È importante sottolineare che questi valori non rappresentano una misura assoluta della correttezza del sistema, ma una stima basata sulla valutazione manuale degli output generati. Per questo motivo, i risultati devono essere interpretati soprattutto come

confronto tra generazione iniziale e refinement, più che come valutazione definitiva della qualità del modello.

Prestazioni

Tipo di chiamata	Media (s)	Mediana (s)	Min (s)	Max (s)
Generazione iniziale	26.815	19.095	4.668	137.907
Refinement	25.182	20.876	4.537	107.838

Tabella 4.6: Statistiche di latenza per tipo di chiamata API.

La Tabella 4.6 riporta, oltre alla media, anche la mediana, il valore minimo e il valore massimo delle latenze registrate. Questa scelta permette di interpretare meglio la distribuzione dei tempi di risposta, poiché alcune chiamate API hanno richiesto un tempo significativamente superiore rispetto alla maggior parte delle esecuzioni.

Nel caso della generazione iniziale, la latenza media è pari a 26.815 secondi, mentre la mediana è 19.095 secondi. La differenza tra questi due valori indica la presenza di alcune chiamate molto più lente, che aumentano la media complessiva. Lo stesso comportamento si osserva nel refinement, dove la media è pari a 25.182 secondi e la mediana a 20.876 secondi.

I valori massimi, pari a 137.907 secondi per la generazione iniziale e 107.838 secondi per il refinement, rappresentano *outliers* rispetto al comportamento più comune del sistema. Questi casi possono essere dovuti a fattori esterni al prototipo, come il carico del provider, il routing della richiesta tramite OpenRouter, la disponibilità temporanea del modello o eventuali rallentamenti nella risposta dell'API.

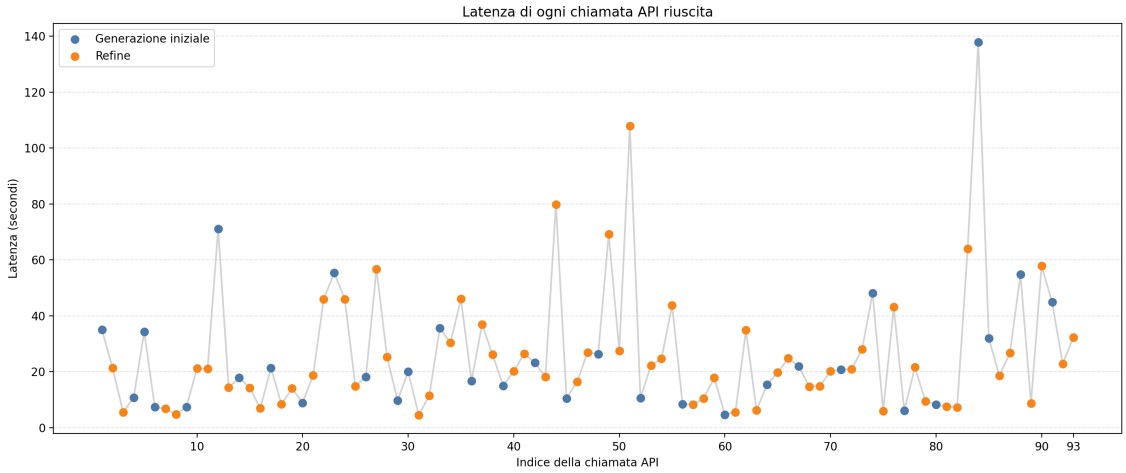


Figura 4.1: Latenza delle chiamate API per singola esecuzione.

La Figura 4.1 mostra la latenza delle singole chiamate API, dove la maggior parte delle chiamate si concentra su valori inferiori rispetto ai massimi riportati in tabella, mentre alcune esecuzioni isolate presentano tempi sensibilmente più elevati. Questi picchi confermano la presenza di outliers e giustificano l'uso della mediana accanto alla media.

Dall'analisi dei log delle chiamate API sono emersi tre stati principali:

- `200 - OK`: la chiamata è stata completata correttamente e il backend ha ricevuto una risposta dal modello. Queste chiamate sono state utilizzate per la valutazione della qualità delle interfacce generate e per il calcolo delle latenze.
- `404 - Not Found`: la chiamata non è stata completata perché il provider o l'endpoint selezionato non era disponibile oppure non supportava il tipo di input inviato. In alcuni casi, ad esempio, il modello selezionato non supportava correttamente l'input con immagine.
- `429 - Too Many Requests`: la chiamata è stata rifiutata a causa di un limite temporaneo di utilizzo o perché il modello risultava momentaneamente occupato. Questo tipo di errore è legato soprattutto all'utilizzo di modelli gratuiti o a disponibilità variabile tramite provider esterni.

Complessivamente, i log contengono sia chiamate completate correttamente sia chiamate terminate con errore. Gli errori API non sono stati considerati come fallimenti qualitativi dell'interfaccia generata, poiché in questi casi il modello non ha prodotto

un output valutabile. Tuttavia, sono rilevanti per valutare l'affidabilità complessiva del sistema e la sua dipendenza dal provider esterno.

Casi principali

Oltre ai risultati quantitativi, è importante evidenziare i principali casi rappresentativi emersi durante la valutazione qualitativa.
Questi casi permettono di osservare in modo più dettagliato il comportamento del sistema con diversi tipi di problemi, scenari e modelli linguistici utilizzati da OpenRouter.

Esempio riuscito

In questo scenario l'utente carica l'immagine di un rubinetto con una perdita visibile, quindi il problema è già abbastanza chiaro dall'input iniziale.

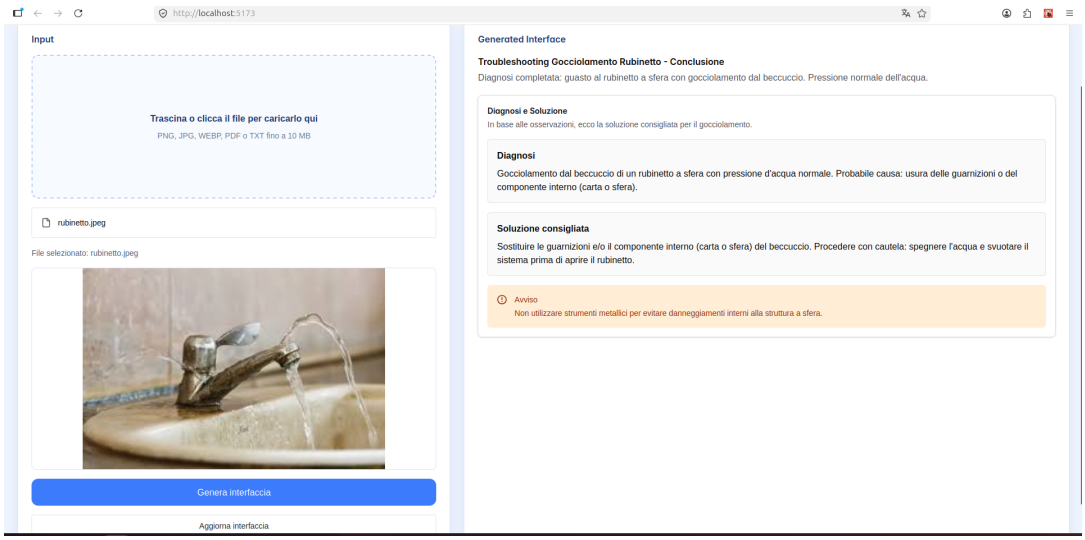


Figura 4.2: Caso riuscito: interfaccia generata per uno scenario con perdita dal rubinetto.

Figura 4.2: L'utente dopo aver caricato l'immagine di un rubinetto con un'evidente perdita d'acqua visibile, riceve da parte del sistema una UI con opzioni specifiche per avere un'ulteriore conferma del problema. L'interfaccia prodotta identifica correttamente il problema e offre all'utente: una diagnosi finale, una soluzione possibile da effettuare e un messaggio di avviso per evidenziare un possibile rischio sull'intervento da eseguire.

Refinement efficace

In questo scenario la prima UI generata è utile ma incompleta, perciò l'utente fornisce un dettaglio aggiuntivo tramite feedback.

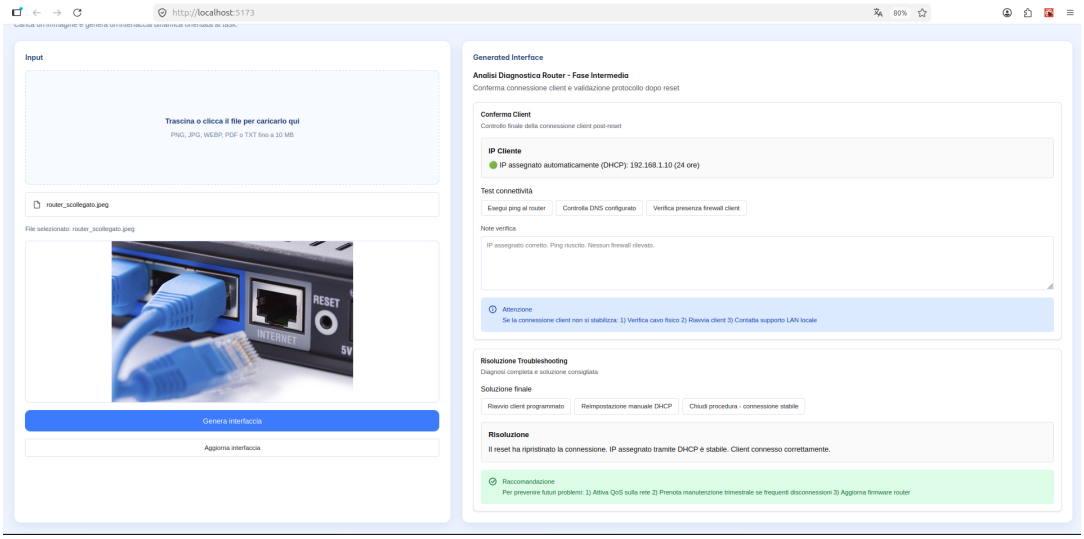


Figura 4.3: Refinement efficace: aggiornamento dell'interfaccia dopo il feedback dell'utente.

Figura 4.3: La prima interfaccia generata forniva una procedura generale per identificare il problema, ma risultava incompleta con alcuni passaggi mancanti rispetto allo scenario presentato. Per questo motivo, è stato necessario aggiungere un dettaglio specifico nella `textarea` della UI, dove a partire da questa informazione, il sistema ha aggiornato la UI aggiungendo passaggi più mirati e correggendo alcune indicazioni precedenti. Il refinement è stato considerato efficace perché non ha ricostruito l'interfaccia da zero, ma ha migliorato la procedura mantenendo sia una label di conferma da parte dell'utente, che una possibile risoluzione e raccomandazione per l'intervento da eseguire.

Parzialmente riuscito

Questo caso rappresenta una situazione in cui il sistema produce una UI renderizzabile, ma non completamente aderente al problema presentato.

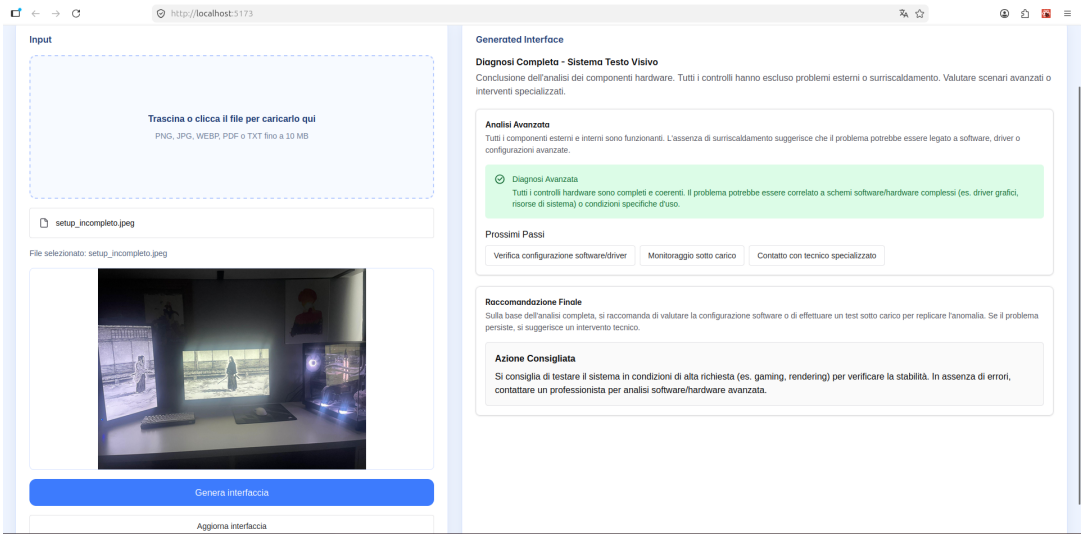


Figura 4.4: Caso parzialmente riuscito: output valido ma non completamente aderente allo scenario.

Figura 4.4: Il sistema genera inizialmente una UI strutturalmente valida, ma il contenuto di alcuni componenti non è corretto o non sufficientemente specifico al problema, perciò il caso è stato classificato come parzialmente riuscito. L'interfaccia di refinement finale può supportare l'utente in una fase iniziale, ma non è sufficiente per guidarlo in modo preciso verso una diagnosi o una soluzione.

Esempio fallito

Questo scenario mostra un caso in cui il modello non riesce a trasformare correttamente l'input iniziale in una UI utile per l'utente.

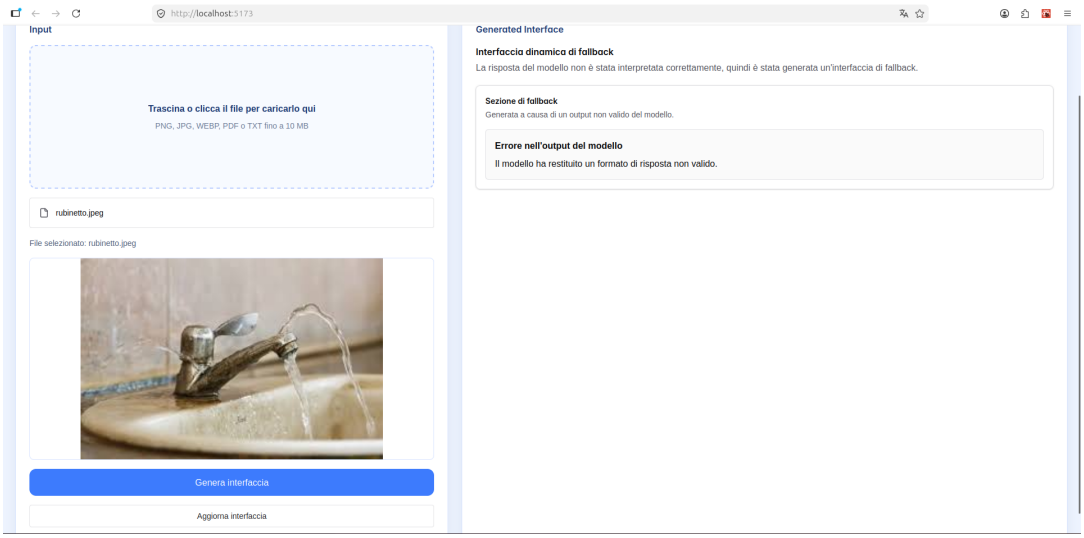


Figura 4.5: Caso fallito: output non utile o non correttamente generato dal modello.

Figura 4.5: Il sistema genera un output strutturalmente non adeguato allo scenario, con componenti non coerenti al problema presentato. Tale errore può dipendere da una risposta incompleta da parte del modello, da un'interpretazione errata del problema oppure da una struttura JSON non renderizzabile. In questo caso l'interfaccia non fornisce indicazioni realmente utili per l'utente e non consente di procedere in modo affidabile nel troubleshooting. Il caso è quindi stato classificato come fallito, poiché il sistema non riesce a trasformare correttamente l'input iniziale in una UI funzionale.

Fallback

Questo caso riguarda l'attivazione del fallback, usato quando il modello non produce un output valido o utilizzabile dal frontend.

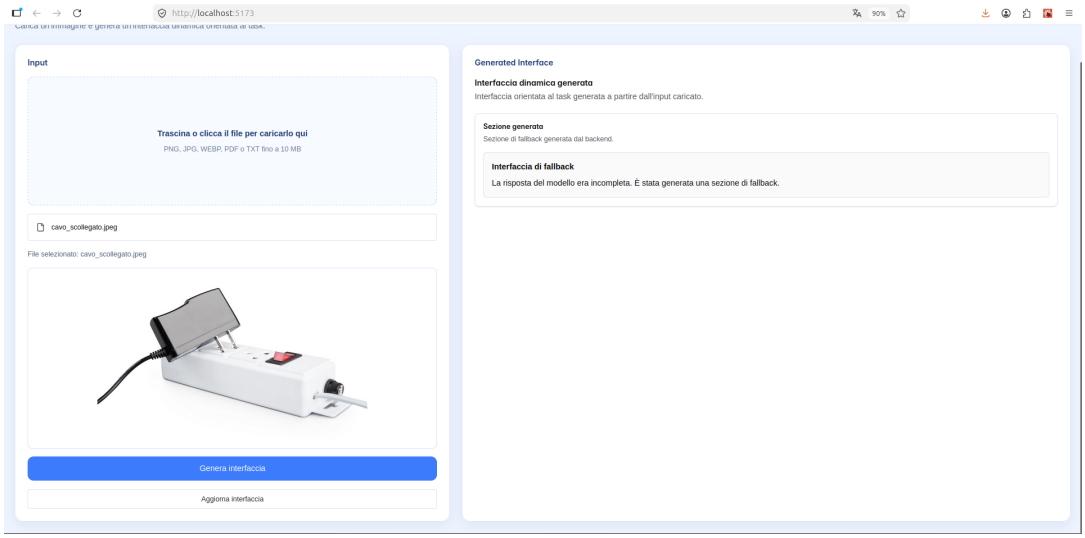


Figura 4.6: Meccanismo di fallback attivato in assenza di un output valido.

Figura 4.6: Il sistema genera una UI di *fallback* nel caso in cui il modello non riesca a generare un output valido. Questo avviene quando l'output non rispetta il formato JSON richiesto da una risposta incompleta del modello, da una struttura JSON non pienamente utilizzabile oppure da un'interpretazione errata del problema. Questo tipo di interfaccia non risolve direttamente il problema, ma permette di ottenere una risposta tale per cui l'utente può comunque evitare un errore di sistema o una schermata vuota.

4.5 Limitazioni e minacce alla validità

La valutazione svolta presenta alcune limitazioni che devono essere considerate nell'interpretazione dei risultati. Innanzitutto, il numero di scenari utilizzati è limitato e non copre tutte le possibili situazioni di troubleshooting fisico. Gli scenari scelti permettono di osservare il comportamento del sistema in casi differenti, sia con problemi visibili sia con problemi più incerti, ma non rappresentano l'intera varietà di situazioni reali. Un secondo limite riguarda la valutazione manuale della qualità funzionale: metriche come l'accuratezza della decomposizione del task e il task success rate dipendono dal confronto tra la UI generata e il comportamento atteso per ciascuno scenario. Anche se i criteri di valutazione sono stati mantenuti coerenti tra i diversi test, rimane comunque una componente soggettiva legata all'interpretazione dell'output prodotto dal modello. Un'altra minaccia alla validità riguarda la dipendenza da modelli LLM/VLM accessibili tramite

OpenRouter. Le prestazioni del sistema possono variare in base al modello selezionato, al provider effettivamente utilizzato, al carico del servizio e alla disponibilità temporanea delle API. Inoltre, alcuni modelli disponibili tramite OpenRouter, soprattutto quelli gratuiti o sperimentali, possono avere un numero limitato di chiamate giornaliere o essere soggetti a rate limit. Questo ha influenzato la quantità di test eseguibili e ha reso più difficile ottenere una distribuzione perfettamente uniforme tra modelli, scenari e fasi di generazione. Anche le misure di latenza devono essere interpretate con cautela, poiché i tempi registrati non dipendono esclusivamente dal prototipo sviluppato, ma anche da fattori esterni come il routing della richiesta, il carico del provider, la disponibilità del modello e possibili rallentamenti della rete.

Considerando questi fatti, i risultati ottenuti devono essere considerati come una valutazione preliminare del prototipo, utile per mostrare la fattibilità dell'approccio e confrontare la fase di generazione iniziale con quella di refinement, ma in alcuni casi può risultare insufficiente per una validazione definitiva in contesti reali o su larga scala.

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi è stato sviluppato un prototipo per la generazione dinamica di interfacce utente basate su LLM/VLM, applicato a scenari di troubleshooting fisico. L'obiettivo principale è stato quello di esplorare un approccio alternativo rispetto alla classica interazione basata su chat, in cui il modello non produce semplicemente una risposta testuale, ma una UI strutturata. In questo modo l'utente può interagire con componenti specifici nella risoluzione di un problema attraverso controlli, domande, suggerimenti e messaggi contestualizzati. Un aspetto chiave del lavoro svolto riguarda il formato dell'output generato dal modello, il quale non genera codice frontend, ma una descrizione astratta dell'interfaccia in formato JSON, composta da sezioni e componenti predefiniti. Tale implementazione permette al frontend di mantenere il controllo sul rendering e riduce il rischio di produrre interfacce non gestibili o incoerenti dal punto di vista tecnico. Inoltre, il processo di refinement consente di aggiornare l'interfaccia sulla base del feedback dell'utente, rendendo la UI non un risultato statico, ma una struttura modificabile durante l'interazione.

La valutazione sperimentale ha mostrato che la generazione iniziale è in grado di produrre una prima interfaccia utile nella maggior parte degli scenari analizzati, anche se non sempre completa o perfettamente aderente al problema. I risultati migliori sono stati ottenuti nella fase di refinement, dove il sistema riceve informazioni aggiuntive e può correggere o completare la UI già generata. Questo conferma l'utilità di un approccio iterativo, soprattutto nei casi in cui il problema non è immediatamente evidente o richiede controlli progressivi da parte dell'utente.

Dai test sono apparsi anche evidenti limiti, come:

- La qualità delle interfacce dipende dal modello utilizzato, dalla chiarezza dell'input e dalla risposta JSON prodotta
- L'utilizzo di provider esterni come OpenRouter introduce variabilità nelle latenze di risposta, possibili errori API e limiti nel numero di chiamate disponibili, soprattutto nel caso dell'utilizzo di modelli gratuiti
- La valutazione manuale della qualità funzionale, seppur basata su criteri ricorrenti, presenta una componente soggettiva che può influenzare i risultati ottenuti

Tra gli sviluppi futuri possibili, si potrebbe considerare l'integrazione di modelli più avanzati o specializzati, in grado di interpretare meglio *input multimodali* e produrre *output multimodali*, in modo da ottenere anche come risposta immagini o video. Questo potrebbe migliorare la qualità delle diagnosi, ridurre gli errori nella generazione del JSON e rendere più fluido il processo di refinement. Inoltre, sarebbe utile valutare l'efficacia del sistema su un numero maggiore di problemi fisici, analizzando casi più complessi, ambigui o realistici.
Infine, un ulteriore sviluppo potrebbe riguardare l'estensione del sistema per una gestione più avanzata dello stato dell'interfaccia, ad esempio tracciando le sezioni già completate, i controlli eseguiti e le decisioni prese dall'utente. Questo permetterebbe di avvicinare il prototipo a un vero assistente procedurale, capace di adattare progressivamente la UI in base all'avanzamento del task.

Appendice A

Codice sorgente significativo

In questa appendice sono riportati gli estratti di codice più rilevanti per comprendere l'implementazione del sistema.

A.1 Backend

Il backend espone due endpoint principali: uno per la generazione iniziale dell'interfaccia e uno per il refinement. Il primo riceve il file caricato dall'utente, crea una sessione e invoca la generazione della UI. Il secondo riceve lo stato corrente dell'interfaccia e il feedback dell'utente, producendo una nuova versione della UI.

```
async def generate_workflow_ui_route(file: UploadFile = File(...)):
    file_info = await save_upload_file(file)
    session_id = create_workflow_session([file_info])
    response_data = await generate_workflow_ui([file_info])
    response_data["session_id"] = session_id

    update_workflow_session(
        session_id=session_id,
        current_ui=response_data,
        feedback_state={}
    )

    return response_data

```

```python

@router.post("/workflow-ui/refine")
async def refine_workflow_ui_route(request: WorkflowRefineRequest):
    session = get_workflow_session(request.session_id)

    file_infos = session.get("files", [])
    response_data = await refine_workflow_ui(
        file_infos=file_infos,
        current_ui=request.current_ui,
        feedback_state=request.feedback_state,
    )

    response_data["session_id"] = request.session_id

    update_workflow_session(
        session_id=request.session_id,
        current_ui=response_data,
        feedback_state=request.feedback_state,
    )

    return response_data
```

Listing A.1: Endpoint principali del backend

A.2 Prompt e integrazione con OpenRouter

Il backend definisce uno schema JSON, costruisce il prompt e richiede al modello utilizzato tramite OpenRouter di produrre un output strutturato e compatibile con il frontend.

```python
WORKFLOW_UI_SCHEMA = {
    "type": "object",
    "properties": {
        "title": {"type": "string"},
        "summary": {"type": "string"},
        "sections": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
```

```
                        "id": {"type": "string"},
                        "title": {"type": "string"},
                        "description": {"type": "string"},
                        "components": {
                            "type": "array",
                            "items": {
                                "type": "object",
                                "properties": {
                                    "component": {
                                        "type": "string",
                                        "enum": [
                                            "checkbox_group",
                                            "radio_group",
                                            "button_group",
                                            "slider",
                                            "select",
                                            "textarea",
                                            "alert",
                                            "info_card",
                                        ],
                                    },
                                },
                                "required": ["component"],
                                "additionalProperties": True,
                            },
                        },
                    },
                    "required": ["id", "title", "description", "
                        components"],
                },
            },
        },
        "required": ["title", "summary", "sections"],
}
```

Listing A.2: Schema JSON atteso per la UI generata

```
payload = {
    "model": settings.OPENROUTER_MODEL,
    "messages": [
        {
            "role": "user",
            "content": content,
```

```
            }
        ],
        "response_format": {
            "type": "json_schema",
            "json_schema": {
                "name": "workflow_ui_response",
                "strict": True,
                "schema": WORKFLOW_UI_SCHEMA,
            },
        },
    }

    headers = {
        "Authorization": f"Bearer {settings.OPENROUTER_API_KEY}",
        "Content-Type": "application/json",
        "HTTP-Referer": "http://localhost:5173",
        "X-Title": "DynamicAI",
    }

    async with httpx.AsyncClient(timeout=90.0) as client:
        response = await client.post(
            settings.OPENROUTER_URL,
            headers=headers,
            json=payload,
        )
```

Listing A.3: Chiamata a OpenRouter con output strutturato

```
prompt = f"""
Hai gia' generato una prima interfaccia strutturata.
Ora devi aggiornarla sulla base del feedback dell'utente.

Interfaccia attuale:
{json.dumps(current_ui, ensure_ascii=False)}

Feedback raccolto finora:
{json.dumps(feedback_state, ensure_ascii=False)}

La nuova interfaccia deve mostrare un avanzamento reale
verso la risoluzione del problema.

Restituisci SOLO JSON valido.
Non trasformare l'interfaccia in una chat.
```

```
16 """
```

Listing A.4: Estratto del prompt di refinement

A.3 Frontend e rendering dinamico

La struttura JSON successivamente generata dal backend, viene trasformata in componenti React.
Ogni valore del campo `component` viene associato a un componente della libreria dinamica.

```
1  import ChecklistBlock from "./register/ChecklistBlock.jsx";
2  import RadioBlock from "./register/RadioBlock.jsx";
3  import ButtonBlock from "./register/ButtonBlock.jsx";
4  import SliderBlock from "./register/SliderBlock.jsx";
5  import SelectBlock from "./register/SelectBlock.jsx";
6  import TextareaBlock from "./register/TextareaBlock.jsx";
7  import AlertBlock from "./register/AlertBlock.jsx";
8  import InfoCardBlock from "./register/InfoCardBlock.jsx";
9
10 export const RegistroComponenti = {
11   checkbox_group: ChecklistBlock,
12   radio_group: RadioBlock,
13   button_group: ButtonBlock,
14   slider: SliderBlock,
15   select: SelectBlock,
16   textarea: TextareaBlock,
17   alert: AlertBlock,
18   info_card: InfoCardBlock,
19 };
```

Listing A.5: Registro dei componenti dinamici del frontend

```
1 {section.components?.map((component, index) => {
2   const ComponentToRender =
3     RegistroComponenti[component.component];
4
5   if (!ComponentToRender) {
6     return (
7       <Box
```

```
          key={`${section.id}-${index}`}
          p={3}
          borderRadius="md"
          bg="red.50"
          border="1px solid"
          borderColor="red.200"
        >
          <Text fontSize="sm" color="red.700">
            Unsupported component: {component.component}
          </Text>
        </Box>
      );
    }

    return (
      <ComponentToRender
        key={`${section.id}-${index}`}
        {...component}
        sectionId={section.id}
        onValueChange={updateSectionFeedback}
      />
    );
  })}
```

Listing A.6: Rendering dinamico dei componenti in Workflow.jsx

```
{generatedUI && !loading && (
  <Workflow
    data={generatedUI}
    feedbackState={feedbackState}
    setFeedbackState={setFeedbackState}
  />
)}
```

Listing A.7: Uso del renderer Workflow nell'interfaccia principale

Appendice B

Repository del progetto

Il codice sorgente completo del progetto è disponibile nel seguente repository GitHub:

`https://github.com/Payam2003/DynamicAI`

Riferimenti bibliografici

[CMW24] Yuzhe Cai, Shaoguang Mao, and Wenshan Wu. Low-code LLM: Graphical user interface over large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: System Demonstrations)*, pages 12–25, 2024. https://aclanthology.org/2024.naacl-demo.2/.

[DWS25] Ian Drosos, Jack Williams, and Advait Sarkar. Dynamic prompt middleware: Contextual prompt refinement controls for comprehension tasks. In *Proceedings of the 4th Annual Symposium on Human-Computer Interaction for Work*, pages 1–23, 2025. https://dl.acm.org/doi/10.1145/3729176.3729203.

[IBM26] IBM. Che cosa sono i modelli linguistici di grandi dimensioni (LLM)?, 2026. https://www.ibm.com/it-it/think/topics/large-language-models. Ultima visita: 20 aprile 2026.

[LXM24] Zelong Li, Shuyuan Xu, and Kai Mei. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821*, 2024. https://arxiv.org/abs/2407.12821.

[Met26] Meta Platforms, Inc. React documentation, 2026. https://react.dev/. Ultima visita: 20 aprile 2026.

[NCW24] Dang Nguyen, Jian Chen, and Yu Wang. Gui agents: A survey. *arXiv preprint arXiv:2412.13501*, 2024. https://arxiv.org/abs/2412.13501.

[OA24] Allard Oelen and Sören Auer. Leveraging large language models for realizing truly intelligent user interfaces. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, pages 1–8, 2024. https://dl.acm.org/doi/10.1145/3613905.3650949.

[Oll26] Ollama. Ollama, 2026. https://ollama.com/. Ultima visita: 29 aprile 2026.

[Ope26] OpenRouter. Openrouter documentation, 2026. https://openrouter.ai/. Ultima visita: 22 aprile 2026.

[Pyt26] Python Software Foundation. Python documentation, 2026. https://www.python.org/. Ultima visita: 27 aprile 2026.

[Ram26] Sebastián Ramírez. Fastapi documentation, 2026. https://fastapi.tiangolo.com/. Ultima visita: 29 aprile 2026.

[Vit26] Vite. Vite documentation, 2026. https://vite.dev/. Ultima visita: 15 aprile 2026.

[Zha24] Chaoyun Zhang. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279*, 2024. https://arxiv.org/abs/2411.18279.

Ringraziamenti

Innanzitutto vorrei ringraziare il Prof. Vitali per avermi seguito durante la dissertazione, per la disponibilità e per i suggerimenti ricevuti per questa tesi. Inoltre, ringrazio anche la Dott.ssa Mengyuan per gli stessi motivi e per il tempo dedicato al confronto sul lavoro svolto.

Grazie a tutti i miei compagni di università: Giangi, Dani, Fillo, Ivan, Lollo, Tambo e Vitto, per tutte le chiacchierate e i momenti passati insieme durante questo percorso. Un ringraziamento speciale va a tutti gli amici di una vita di Anzola, ma anche ai ragazzi della barca, della croce e alle girls di Apericolos. Vi ringrazio per avermi sempre supportato e sopportato, per non avermi mai lasciato solo nei momenti più difficili e per tutti i momenti di divertimento e spensieratezza che abbiamo condiviso.

Infine, il ringraziamento più grande va alla mia famiglia. Ringrazio i miei genitori e mio fratello per avermi sempre supportato in tutte le mie scelte, per essermi stati vicini nei momenti più difficili e per aver creduto in me durante tutto questo percorso. Questo traguardo è anche loro.