Università degli Studi di Napoli "Parthenope"
Dipartimento di Scienze e Tecnologie
Corso di laurea in Informatica

Elaborato finale di Laurea

Sviluppo di un ambiente virtuale per ridurre il senso di solitudine nella popolazione anziana

Integrazione di agenti virtuali per il sostegno alla socialità

Relatore
Prof.ssa Paola Barra

Candidato
Luca Tartaglia
Matr. 0124002294

Anno Accademico 2024/2025

ABSTRACT

In questa tesi presento **SOULFRAME**, un sistema conversazionale con avatar 3D pensato per rendere l'interazione con l'AI più semplice, naturale e accessibile. L'obiettivo principale è costruire un'esperienza d'uso intuitiva: l'utente deve poter parlare con l'avatar senza complessità tecniche, usando un'interfaccia adattabile sia a desktop (controlli da tastiera) sia a mobile/touch (push-to-talk e navigazione semplificata).

Il progetto è stato sviluppato con un'architettura modulare client-server. Sul lato frontend, **Unity WebGL** gestisce flusso UI, avatar e acquisizione audio. Sul lato backend, microservizi **FastAPI** si occupano di trascrizione vocale con **Whisper (STT)**, risposta contestuale con **LLM + RAG** (Ollama e ChromaDB), sintesi vocale con **Coqui XTTS v2 (TTS)** e gestione/caching degli asset avatar. La memoria è persistente per singolo avatar e può essere arricchita con note, documenti PDF e immagini, anche tramite OCR.

Una parte importante del lavoro riguarda l'automazione operativa: sono stati introdotti script di setup e gestione servizi per installazione e deploy in modo rapido su Windows e Ubuntu, riducendo errori manuali e tempi di configurazione. Durante lo sviluppo sono state affrontate criticità di integrazione tra servizi, latenza end-to-end e compatibilità tra ambienti. I risultati ottenuti mostrano un sistema funzionante, estendibile e sufficientemente robusto per conversazioni vocali contestuali in scenari realistici.

INDICE

1. INTRODUZIONE

1.1 Motivazione e contesto applicativo

La realtà virtuale (VR) si distingue dagli altri media interattivi per la capacità di far sentire l'utente presente in un luogo diverso da quello fisico. Slater e Sanchez-Vives descrivono questa presenza come il risultato di due fattori principali: la *place illusion*, legata alla coerenza tra movimenti e scena virtuale, e la plausibilità degli eventi, cioè quanto l'ambiente reagisce in modo credibile alle azioni dell'utente [**Slater2016VR**]. Per questo, non basta una buona qualità visiva. Conta anche come il sistema risponde e quanto le interazioni risultano significative.

In questo quadro, la VR viene usata da tempo per training, formazione e simulazioni in ambito medico e professionale, soprattutto quando serve riprodurre situazioni costose, rischiose o difficili da standardizzare. Diversi studi segnalano anche l'interesse per contesti con forte componente sociale, dove realismo e naturalezza dell'interazione influenzano direttamente l'esperienza.

Il limite emerge quando l'ambiente resta statico o gli attori virtuali seguono script rigidi. In questi casi l'interazione si riduce a scelte predefinite e perde adattività. In uno studio sul training in VR, Kan, Rumpelnik e Kaufmann confrontano agenti conversazionali con agenti a audio preregistrato. I risultati mostrano che una pipeline vocale con riconoscimento del parlato e sintesi vocale aumenta in modo significativo la co-presenza percepita [**Kan2023ECA__Training**]. Questo indica che il dialogo non è un elemento accessorio, ma una parte centrale dell'efficacia del sistema.

Gli Embodied Conversational Agents (ECA), cioè agenti conversazionali con corpo virtuale e segnali multimodali, nascono proprio da questa esigenza. La revisione sistematica di Yang e coautori mostra che la maggior parte degli studi in XR si concentra sulla VR, spesso con HMD e con Unity come piattaforma princi-

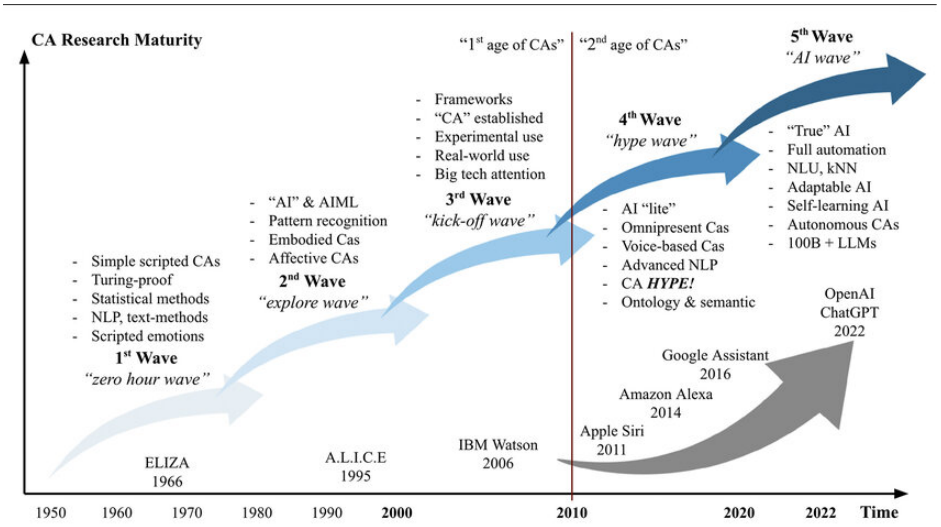

Fig. 1.1: Evoluzione della maturità della ricerca sui Conversational Agents (CA), dalla "zero hour wave" alla "AI wave", con riferimento ai principali passaggi tecnologici. [1]

pale [**Yang2025ECA_XR**]. La stessa revisione evidenzia che molte interazioni restano *task-oriented*, ma cresce l'interesse per scambi più dinamici e adattivi grazie a modelli neurali e, più recentemente, ai *Large Language Models* (LLM). In SOULFRAME, questo filone viene adattato a una configurazione senza HMD, centrata sulla generazione e animazione di avatar a partire da segnali visivi e vocali dell'utente.

Dal punto di vista dei canali, la voce è ormai una scelta frequente sia in input sia in output. Resta però aperta la sfida di integrare dialogo libero e contestuale con un embodiment coerente.

Figura 1.1 sintetizza l'evoluzione dei Conversational Agents (CA) dalla fase iniziale di sistemi scriptati fino all'attuale ondata guidata da modelli linguistici avanzati. In questa traiettoria, l'integrazione di linguaggio naturale, adattività e autonomia rende più rilevante il tema dell'*embodiment* in ambienti immersivi. In questo quadro, SOULFRAME non nasce come semplice demo grafica: punta a collegare presenza sociale e interazione credibile in un ambiente 3D familiare, con la conversazione vocale come asse centrale.

SOULFRAME si inserisce in questo contesto come sistema immersivo con

[1] Fonte: adattamento da una figura pubblicata su ResearchGate.

ECA vocale embodied. L'obiettivo è testare una pipeline completa che integra voce e un profilo avatar creato in fase iniziale tramite acquisizione visiva dell'utente, così da replicarne aspetto e stile comportamentale durante il dialogo. Un punto centrale è la memoria dell'agente: il sistema combina contesto conversazionale e recupero di conoscenza (*RAG*) per generare risposte coerenti con quanto detto e acquisito durante l'interazione. L'ipotesi di fondo è semplice: in ambienti immersivi e familiari, la qualità percepita dipende anche dalla capacità dell'agente di sostenere scambi plausibili, rapidi e contestualmente informati.

1.2 Perimetro e requisiti di progetto

SOULFRAME è un prototipo di interazione immersiva con un ECA vocale embodied in ambiente 3D. Lo scopo è valutare fattibilità tecnica e qualità percepita dell'interazione conversazionale in scenari con componente sociale. Il progetto non vuole essere una piattaforma generale per creare mondi virtuali e non è una soluzione clinica certificata. Questi obiettivi richiedono validazioni regolatorie e studi longitudinali fuori dal perimetro di una tesi triennale.

Il perimetro si concentra quindi sull'integrazione *end-to-end* della pipeline vocale, visiva e dialogica dentro una scena immersiva, con coerenza tra risposta verbale e comportamenti *embodied* dell'agente.

Sul piano tecnologico, il sistema integrato di SOULFRAME usa la fotocamera soprattutto nella fase di configurazione iniziale dell'utente, per creare e salvare il profilo avatar; durante l'interazione ordinaria il sistema si basa principalmente su microfono, memoria di contesto e stato conversazionale. La resa della scena avviene con moduli di generazione/animazione avatar e con un motore real-time 3D. Questa scelta favorisce replicabilità, facilita sostituzioni tra componenti e sostiene un'interazione più familiare, orientata alla replica visiva e comportamentale della persona.

I requisiti funzionali principali derivano dalla necessità di supportare un dialogo naturale in tempo quasi reale. L'input vocale è gestito con logica *push-to-talk*: l'utente tiene premuto un tasto per parlare e rilascia per chiudere il turno. A quel punto il sistema trascrive l'audio con Speech-to-Text (STT) e genera la risposta tramite LLM supportato dalla memoria RAG. La comprensione dell'input emerge dalla combinazione tra modello linguistico, contesto conversazionale e semplici

regole applicative. Il testo prodotto viene poi convertito in audio tramite Text-to-Speech (TTS) e restituito attraverso la voce dell'agente. Nella fase di onboarding, il sistema usa la fotocamera per costruire il profilo visivo dell'avatar, che viene poi riutilizzato durante la conversazione. A questa catena si aggiungono tre funzioni chiave: memoria contestuale con *Retrieval-Augmented Generation* (RAG), descrizione semantica delle immagini per estrarre informazioni dall'ambiente, e acquisizione testuale da documenti tramite OCR, così che l'avatar possa usare anche contenuti visivi e documentali nella conversazione.

I requisiti non funzionali riguardano soprattutto latenza, robustezza e modularità. Per mantenere plausibilità conversazionale, la catena STT–RAG/LLM–TTS deve restare reattiva, anche quando entrano in gioco recupero in memoria, analisi delle immagini e OCR. L'architettura deve anche tollerare guasti parziali senza interrompere l'esperienza immersiva. La modularità è perseguita soprattutto a livello di servizi e interfacce HTTP: i componenti backend possono essere sostituiti in modo relativamente rapido, purché restino compatibili con gli endpoint e i formati attesi dal client.

Per aspetti come la latenza percepita e i criteri di accettabilità, non c'è una soglia unica valida per tutti i contesti applicativi. In questo lavoro i vincoli sono quindi definiti in modo operativo sul prototipo e verificati nei capitoli successivi.

1.3 Obiettivi e contributi di SOULFRAME

Gli obiettivi di SOULFRAME seguono il filone degli studi sugli ECA in XR. Gli studi mostrano una forte presenza di implementazioni VR in Unity, interazioni spesso orientate al compito e una notevole eterogeneità in input, output e metodi di valutazione. Nello stesso tempo, i canali vocali sono molto usati, soprattutto con TTS in uscita [**Yang2025ECA_XR**]. In questo scenario, il progetto punta a costruire un prototipo che renda verificabile l'integrazione *end-to-end* della pipeline vocale con un agente *embodied*, documentando in modo chiaro le scelte tecniche e sperimentali.

Il primo obiettivo (RQ1) riguarda la fattibilità tecnica di un'interazione vocale in tempo reale con un agente embodied in ambiente immersivo 3D. In termini operativi, bisogna dimostrare che la catena STT $\rightarrow$ RAG/LLM $\rightarrow$ TTS $\rightarrow$ output audiovisivo dell'avatar funzioni in modo continuo e stabile, includendo anche

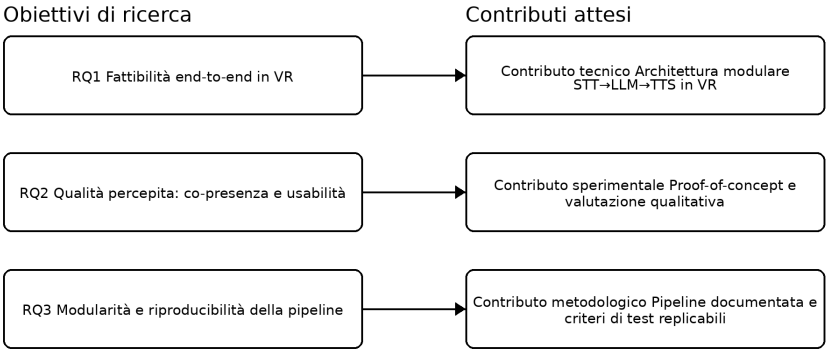

Fig. 1.2: Corrispondenza tra obiettivi di ricerca e contributi del progetto SOULFRAME.

descrizione immagini e OCR documentale. La valutazione considera tempi di elaborazione per blocco, completamento dei turni senza errori bloccanti e gestione corretta delle transizioni tra ascolto, elaborazione e risposta. La fattibilità include anche la coerenza dell'embodiment, cioè un avatar che allinei voce, aspetto visivo e segnali comportamentali.

Il secondo obiettivo (RQ2) riguarda la qualità percepita dell'interazione, in particolare co-presenza e naturalezza dialogica. SOULFRAME prevede almeno una valutazione qualitativa guidata, affiancata quando possibile da metriche soggettive usate negli studi VR con ECA: presenza, co-presenza, qualità dell'interazione e qualità della presentazione delle informazioni, oltre a misure di processo come durata dei compiti e performance percepita. Studi comparativi tra agenti conversazionali e audio pre-scriptato mostrano differenze osservabili, soprattutto sulla co-presenza [**Kan2023ECA__Training**].

Il terzo obiettivo (RQ3) riguarda modularità e riproducibilità della pipeline. In pratica, il sistema deve permettere la sostituzione dei componenti principali senza riscrivere l'intera architettura e deve rendere tracciabili configurazioni, tempi e risultati delle prove. Questa sostituzione avviene mantenendo stabili i contratti API tra client e servizi. Questo obiettivo completa i primi due, perché rende il prototipo confrontabile nel tempo e riutilizzabile in test successivi. Figura 1.2 sintetizza la corrispondenza tra i tre obiettivi e i contributi attesi del progetto.

I contributi si distribuiscono su tre livelli. Il contributo tecnico è un'architettura modulare *end-to-end* basata su microservizi (STT, RAG/LLM, TTS e servizi

di memoria) con interfacce API esplicite e configurabili. Questo consente di sostituire i componenti con impatto limitato sul resto del sistema, a condizione di mantenere compatibilità dei contratti di scambio. Il contributo sperimentale è un *proof-of-concept* verificabile con un set minimo di misure soggettive e osservazioni qualitative, utile a stimare comprensibilità, fluidità e credibilità dell'interazione. Il contributo metodologico è la documentazione riproducibile delle scelte progettuali, con criteri di logging, tracciamento dei tempi e selezione motivata delle misure di *user experience*. Questa impostazione è coerente con i principali lavori sul *dialogue management*, che distinguono approcci *finite-state*, *frame-based* e *agent-based* e raccomandano maggiore standardizzazione nelle metriche tecniche e nella valutazione dell'esperienza utente [**Laranjo2018CA**].

1.4 Panoramica del sistema e flusso end-to-end

SOULFRAME adotta un'architettura client–server pensata per supportare dialogo vocale naturale in un ambiente immersivo 3D. Il client gestisce rendering della scena, rappresentazione dell'agente embodied tramite avatar animato e acquisizione dei segnali utente. La cattura audio non è continua: parte quando l'utente tiene premuto il comando *push-to-talk* e termina al rilascio. La cattura video da fotocamera è invece usata in fase iniziale per configurare e salvare il profilo avatar. Il backend mantiene lo stato conversazionale e orchestra la pipeline linguistica e cognitiva: gestione del contesto, recupero di informazioni con *RAG*, generazione con LLM e integrazione di descrizioni di immagini e testi estratti via OCR. Questa separazione permette di tenere sul client le funzioni sensibili al frame-rate e sul backend i moduli più onerosi e soggetti a evoluzione.

Il flusso end-to-end parte dalla voce dell'utente e ritorna alla risposta audiovisiva dell'avatar. Dopo il rilascio del tasto *push-to-talk*, l'audio viene trascritto dal modulo STT. Il testo viene poi inviato al servizio di chat del backend, che funge da orchestratore RAG/LLM. Quando la memoria dell'avatar è presente, il servizio prova a recuperare contesto rilevante con ricerca ibrida; quando non ci sono ricordi utili, la risposta viene generata senza supporto documentale aggiuntivo. Il contesto può essere arricchito con descrizioni delle immagini e con testo proveniente da documenti acquisiti via OCR. Su questa base, il LLM produce la risposta, che viene sintetizzata dal TTS. In parallelo, il client anima l'avatar

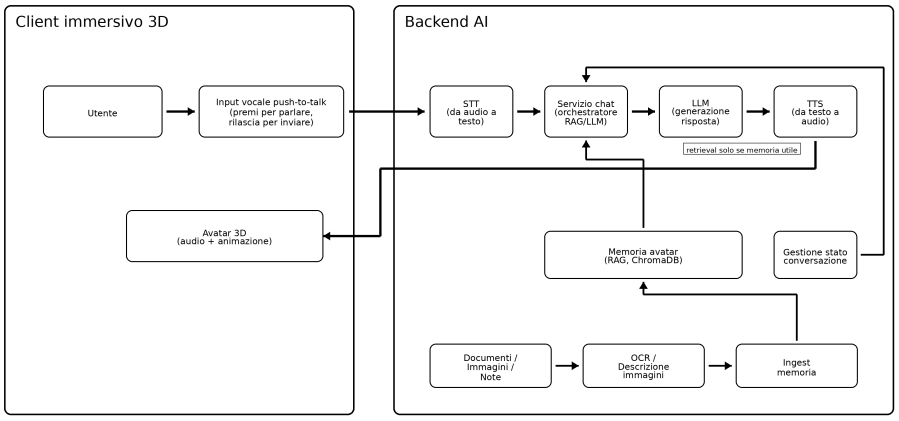

Fig. 1.3: Flusso end-to-end del sistema SOULFRAME: dall'input vocale dell'utente alla risposta dell'agente embodied.

usando il profilo visivo creato in onboarding e regole di comportamento coerenti con il contesto dialogico, così da mantenere allineamento tra contenuto verbale e resa visiva.

La latenza totale dipende dalla somma delle latenze dei singoli moduli e dalla gestione a turni del *push-to-talk*. L'obiettivo non è eliminare ogni variabilità, ma mantenere continuità tra fine enunciato e risposta dell'agente con tempi percepiti stabili. Il backend coordina gli stati operativi e gestisce interruzioni o riformulazioni dell'utente senza perdere coerenza.

La modularità è un principio guida, ma nel senso operativo del progetto: i servizi sono separati, indirizzabili e configurabili, e possono essere sostituiti mantenendo coerenti endpoint e payload. In questo modo si possono confrontare varianti senza riprogettare tutto il sistema client. Figura 1.3 riassume la catena di elaborazione e la separazione di responsabilità tra client e backend.

1.5 Metodo di lavoro e struttura della tesi

Lo sviluppo di SOULFRAME segue un approccio iterativo basato su prototipazione incrementale. La scelta serve a ridurre il rischio tecnico tipico dei sistemi che combinano vincoli di reattività in interazione a turni (*push-to-talk*), elaborazione linguistica e interazione immersiva. Il lavoro procede per integrazioni successive: prima si valida ogni modulo in isolamento, poi si integra la pipeline

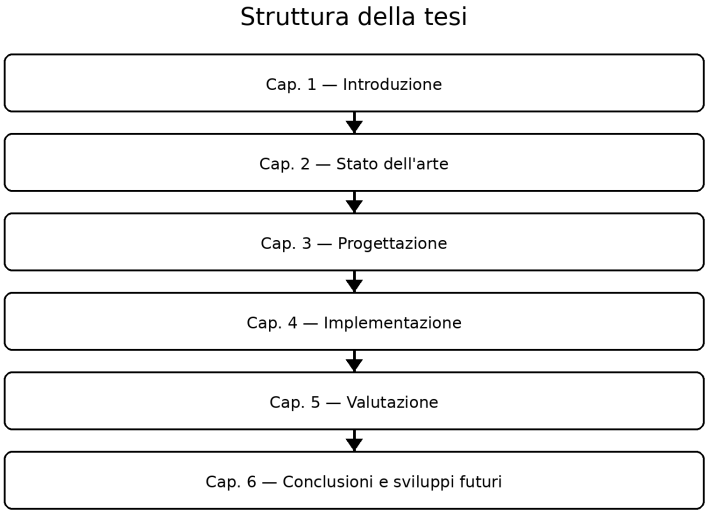

Fig. 1.4: Struttura della tesi e organizzazione dei capitoli.

completa e si verifica il comportamento in scenari via via più complessi. Questo metodo rende più facile individuare colli di bottiglia e problemi di sincronizzazione in fase precoce. Durante tutto il processo vengono tracciate scelte, compromessi e dipendenze per mantenere l'evoluzione del sistema leggibile e replicabile.

La tesi è organizzata in sei capitoli. Il Capitolo 1 introduce problema, perimetro e obiettivi e fornisce la visione d'insieme del sistema. Il Capitolo 2 presenta fondamenti e stato dell'arte su VR, ECA vocali e approcci di memoria conversazionale come RAG, includendo OCR e descrizione immagini come fonti di contesto. Il Capitolo 3 descrive l'architettura di SOULFRAME, i macro-componenti e i criteri progettuali. Il Capitolo 4 entra nelle scelte implementative e tecnologiche, inclusa l'integrazione a microservizi tramite API, la gestione dello stato conversazionale e la resa embodied dell'agente. Il Capitolo 5 riporta test e valutazione, con verifica funzionale della pipeline e stima della qualità percepita. Il Capitolo 6 conclude il lavoro, discute limiti del prototipo e propone sviluppi futuri.

Figura 1.4 offre una vista sintetica della struttura e della progressione logica tra capitoli.

2. FONDAMENTI E STATO DELL'ARTE

2.1 Agenti conversazionali embodied in XR

Gli Embodied Conversational Agents (ECA) sono agenti conversazionali dotati di una rappresentazione corporea, progettati per essere percepiti come interlocutori nello spazio di interazione. In Extended Reality (XR), il corpo virtuale viene collocato in una scena tridimensionale e coordinato con i turni del dialogo, con l'obiettivo di rendere l'interazione più naturale rispetto a un'interfaccia solo testuale.

La letteratura recente mostra che molti sistemi ECA in XR sono sviluppati in Virtual Reality (VR), spesso con Head-Mounted Display (HMD) e con Unity come piattaforma ricorrente. Risulta frequente anche l'uso dell'interazione vocale e di configurazioni uno-a-uno. Molte applicazioni restano orientate a compiti specifici, ma cresce l'interesse verso dialoghi più adattivi supportati da modelli neurali.[**Yang2025ECA_XR**] SOULFRAME si colloca in questo scenario come ECA vocale embodied in un ambiente 3D fruibile via WebGL senza HMD: l'agente conversa in modalità push-to-talk e mantiene un profilo avatar che integra aspetto e voce.

2.1.1 Presenza sociale, co-presenza e ruolo della voce

Quando l'interazione avviene in un ambiente mediato, la presenza sociale descrive la percezione dell'altro come interlocutore con cui si condivide un'esperienza. Una trattazione utile per la valutazione scompone questo fenomeno in componenti distinguibili, includendo la co-presenza (percezione di condividere lo stesso spazio) e forme di coinvolgimento attentivo e comportamentale che rendono il dialogo più contingente e reciproco.[**Oh2018SocialPresence**]

La voce è un canale rilevante per la presenza sociale perché rende immediatamente percepibili tempi di risposta, ritmo e prosodia. In VR questo

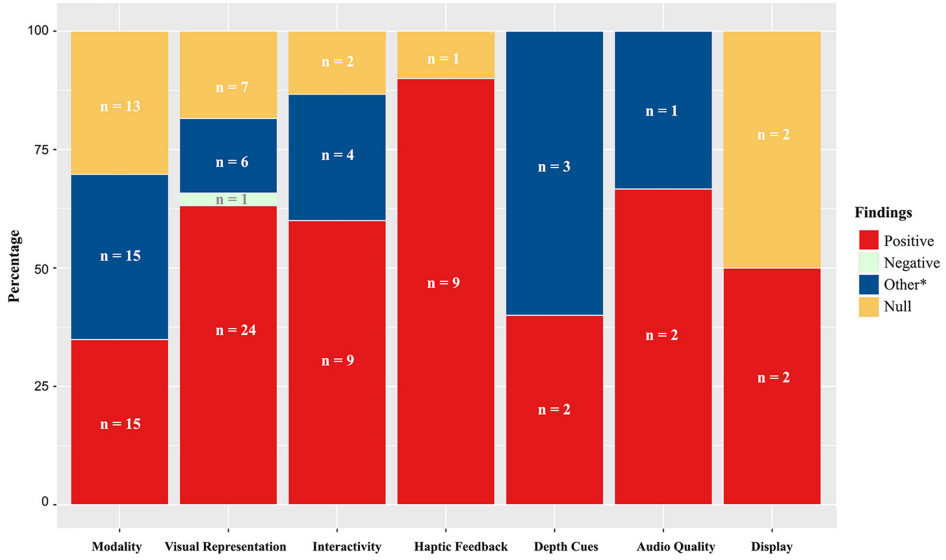

Fig. 2.1: Fattori immersivi associati alla presenza sociale: la qualità audio compare tra le variabili considerate nella letteratura. [1]

effetto risulta più marcato quando l'agente è embodied: evidenze sperimentali indicano che un ECA con interazione vocale in tempo reale (STT+TTS) aumenta la co-presenza percepita rispetto a una condizione con audio pre-registrato.[**Kan2023ECA_Training**] In SOULFRAME, la scelta del push-to-talk e l'uso di una voce persistente per avatar seguono questa logica, con l'obiettivo di stabilizzare i turni e mantenere coerenza d'identità.

La presenza sociale e la co-presenza dipendono dall'allineamento tra canali comunicativi e tempi dell'interazione. Figura 2.1 mostra che la qualità audio è una delle variabili considerate insieme ad altri fattori immersivi. Per un ECA vocale, qualità della voce e gestione temporale dei turni influenzano la naturalezza percepita; in SOULFRAME, push-to-talk, profilo vocale per avatar, warmup e frasi di attesa sono adottati per mitigare sovrapposizioni e silenzi durante l'elaborazione.

2.1.2 Limiti aperti nei sistemi conversazionali immersivi

L'integrazione del dialogo in ambienti 3D introduce criticità che emergono meno nei sistemi testuali tradizionali. La principale è la latenza end-to-end tra

[1] Fonte: Oh et al., *A Systematic Review of Social Presence: Definition, Antecedents, and Implications*, Frontiers in Robotics and AI (2018), https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2018.00114/full (Figura 3; licenza/uso: CC BY).

Fig. 2.2: Schema della pipeline conversazionale (STT → RAG/LLM → TTS) adottata in SOULFRAME.

acquisizione audio, trascrizione, generazione e sintesi: ritardi percepibili riducono la naturalezza percepita anche quando il contenuto della risposta è corretto.

Un secondo limite riguarda la continuità tra turni: senza una memoria affidabile il sistema fatica a mantenere riferimenti e preferenze espresse dall'utente. Un ulteriore nodo è l'integrazione multimodale, che richiede coerenza tra risposta linguistica, tempi e segnali non verbali. Infine, la valutazione resta complessa perché combina metriche soggettive (presenza, co-presenza, naturalezza) e metriche tecniche (latenza, errori di trascrizione), con protocolli non sempre uniformi.

In letteratura, architetture modulari open-source per ECA vocali in VR mostrano che la separazione in servizi STT e TTS semplifica l'integrazione ma rende evidenti i colli di bottiglia temporali, richiedendo strategie come output in streaming.[**Yin2023VoiceVR**] SOULFRAME adotta una pipeline a tre stadi, memoria persistente per avatar e frasi di attesa per ridurre l'impatto dei tempi morti senza dipendenze cloud.

2.2 Pipeline AI adottata in SOULFRAME

SOULFRAME implementa una pipeline conversazionale in tre stadi: riconoscimento del parlato, generazione contestuale e sintesi vocale. Questa scomposizione consente di isolare i vincoli della conversazione a turni, in particolare la latenza, e di aggiornare i singoli moduli senza modificare l'intera architettura. Figura 2.2 mostra il flusso dei dati dal segnale audio in ingresso al testo trascritto, fino al testo di risposta e all'audio sintetizzato.

2.2.1 Speech-to-Text con Whisper

Il primo stadio della pipeline è il riconoscimento automatico del parlato. Nel prototipo, l'audio viene acquisito in push-to-talk dal client WebGL e inviato a un micro-servizio dedicato che restituisce la trascrizione da inoltrare al modulo di memoria e generazione. L'esecuzione locale riduce dipendenze di rete e supporta requisiti di privacy.

SOULFRAME utilizza Whisper, modello STT basato su architettura Transformer e addestrato su larga scala con supervisione debole. L'addestramento su circa 680.000 ore e l'impostazione multilingue (99 lingue) supportano buone prestazioni zero-shot e robustezza a variabilità di parlanti e condizioni acustiche.[**Radford2023Whisper**] Nel prototipo è adottata la versione `medium` come compromesso tra qualità della trascrizione e costo computazionale.

2.2.2 Memoria conversazionale con RAG (LLM + embeddings + retrieval)

Il secondo stadio introduce una memoria conversazionale che combina generazione e recupero di contesto.

In termini operativi, la query testuale viene trasformata in embedding, usata per recuperare dall'indice i passaggi pertinenti e reinserita nel contesto fornito al Large Language Model (LLM). In questo modo la risposta può mantenere continuità tra turni e riutilizzare informazioni già emerse nella conversazione.

L'approccio RAG integra memoria parametrica del modello e memoria non parametrica aggiornata via indice; la conoscenza può quindi evolvere intervenendo sui contenuti recuperabili senza riaddestrare i pesi.[**Lewis2020RAG**] Figura 2.3 mostra il flusso retrieval→generation.

Nel prototipo SOULFRAME la memoria è persistente tra sessioni e separata per avatar, per ridurre interferenze tra profili. Il servizio RAG integra LLM via Ollama, modello di embedding e ChromaDB; supporta note testuali e ingestione di documenti (con OCR per PDF) e adotta retrieval ibrido, combinando similarità semantica e segnali lessicali.

[2] Fonte: Kaur et al., *Knowledge, context and personalization in retrieval-augmented generation for healthcare*, Frontiers in Artificial Intelligence (2025), https://www.frontiersin.org/journals/artificial-intelligence/articles/10.3389/frai.2025.1697169/full (Figura 1; licenza/uso: CC BY).

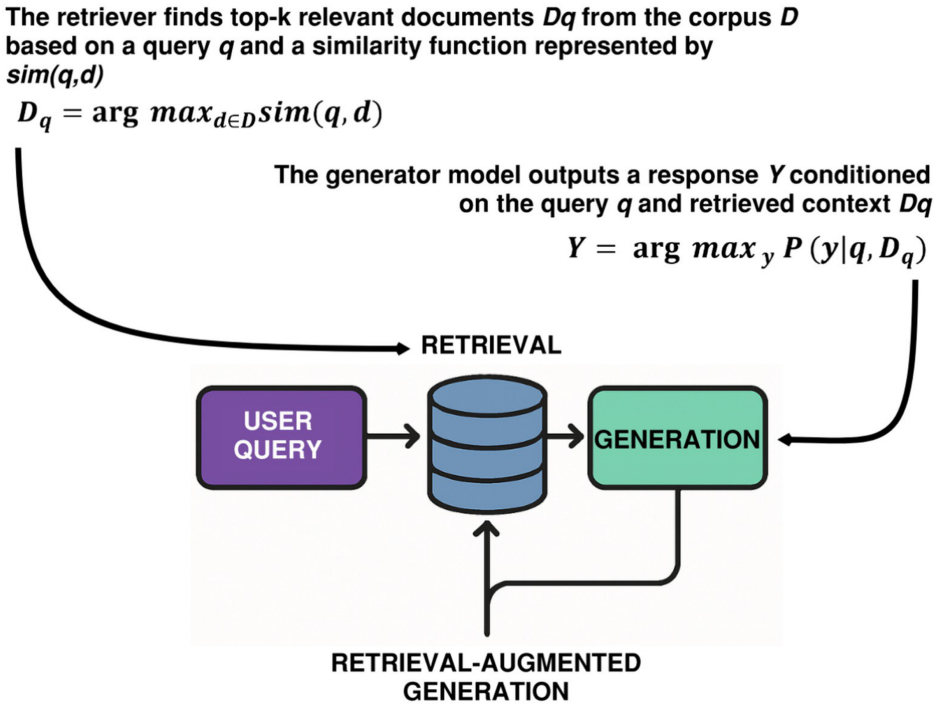

Fig. 2.3: Schema concettuale di Retrieval-Augmented Generation (RAG): recupero di contesto e generazione della risposta. [2]

2.2.3 Text-to-Speech con Coqui XTTS v2

Il terzo stadio converte il testo in audio e chiude il ciclo conversazionale. In un ECA embodied, la sintesi vocale influisce sulla percezione di continuità e coerenza dell'interlocutore.

SOULFRAME adotta Coqui XTTS v2 per supporto multilingue e voice cloning tramite campione di riferimento, senza addestrare un modello vocale dedicato per ogni utente.[**Casanova2024XTTS**] Nel prototipo, il campione è validato rispetto al testo atteso e associato al profilo avatar; il servizio supporta sintesi in streaming, warmup e frasi di attesa per mitigare la latenza percepita.

2.3 Integrazione client-server del sistema

L'architettura di SOULFRAME separa il client, responsabile di rendering e interazione, dal backend, responsabile dell'inferenza AI. Questa scelta riduce il carico sul runtime WebGL e permette di gestire i moduli AI come servizi indi-

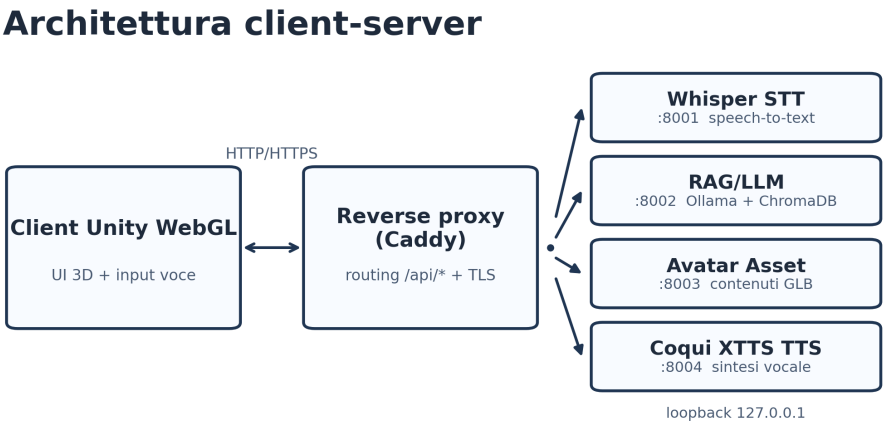

Fig. 2.4: Architettura logica client-server: il frontend Unity WebGL comunica con i servizi applicativi tramite reverse proxy.

pendenti. Figura 2.4 mostra il ruolo del reverse proxy come punto di accesso alle API e l'instradamento verso i micro-servizi dedicati.

2.3.1 Frontend Unity WebGL e principali vincoli di piattaforma

Il frontend è una build Unity WebGL eseguita nel browser. Questa scelta favorisce l'accessibilità senza installazione locale, ma introduce vincoli di piattaforma legati a WebAssembly e al main thread del browser.[3] In SOULFRAME, il client gestisce interfaccia e turni conversazionali, mentre STT, RAG/LLM e TTS sono delegati ai servizi backend.

La modalità push-to-talk riduce sovrapposizioni nei turni e semplifica la gestione dei permessi audio in ambiente web. Il client integra inoltre indicatori di stato e schermate di caricamento per rendere esplicite le fasi di inizializzazione.

2.3.2 Backend FastAPI a micro-servizi

Il backend è implementato in Python e organizzato in micro-servizi FastAPI, ciascuno dedicato a un sottocompito della pipeline. Questa separazione facilita isolamento delle dipendenze, gestione delle risorse e diagnosi dei guasti.

[3] Doc ufficiale: Unity WebGL Technical Limitations, https://docs.unity3d.com/6000.2/Documentation/Manual/webgl-technical-overview.html.

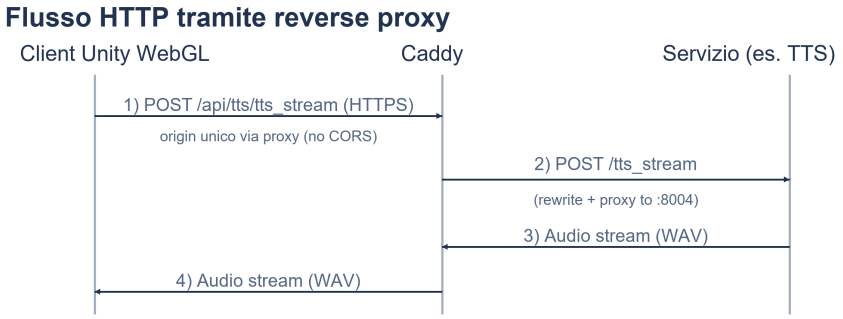

Fig. 2.5: Flusso HTTP semplificato tramite reverse proxy: dal client Unity WebGL al micro-servizio e ritorno dello stream audio.

In sviluppo i servizi sono avviati con `uvicorn`; in esercizio possono essere gestiti in modo indipendente con log separati. FastAPI fornisce interfacce HTTP coerenti e documentazione OpenAPI utile per il collaudo degli endpoint.

2.3.3 Comunicazione HTTP, CORS e proxy applicativo

La comunicazione tra client e backend avviene tramite richieste HTTP con payload testuali e audio. In ambiente browser, la Same-Origin Policy impone vincoli di origine; quando frontend e servizi risiedono su host o porte diverse, è necessaria una configurazione esplicita del CORS tramite FastAPI.[4] Figura 2.5 mostra una richiesta tipica instradata dal reverse proxy verso un micro-servizio e il ritorno dello stream audio.

In SOULFRAME, il reverse proxy termina TLS e riscrive i path applicativi (`/api/*`) verso i servizi interni. Il client comunica quindi con un unico origin HTTPS, mentre le porte dei micro-servizi restano non esposte; questa configurazione riduce problemi di integrazione lato browser e migliora la stabilità della catena di chiamata.

[4] Doc ufficiale: FastAPI CORS, https://fastapi.tiangolo.com/tutorial/cors/.

3. ARCHITETTURA E TECNOLOGIE UTILIZZATE

3.1 Requisiti del sistema

Un prototipo di agente conversazionale embodied in Virtual Reality (VR) combina interazione in tempo reale, gestione di asset 3D e servizi di inferenza esterni. Separare requisiti funzionali e non funzionali distingue cosa il sistema deve fare (operazioni osservabili) dai vincoli che rendono l'esperienza stabile e credibile (latenza, disponibilitÃfÂ , modularitÃfÂ).

3.1.1 Requisiti funzionali

Nel prototipo i servizi Speech-to-Text (STT), Retrieval-Augmented Generation (RAG), Text-to-Speech (TTS) e gestione asset avatar sono esposti come micro-servizi FastAPI su porte dedicate (8001–8004) e invocati dal client Unity WebGL tramite endpoint HTTP.

RF1 (Libreria e selezione avatar): il client deve ottenere e visualizzare la lista degli avatar disponibili tramite `/avatars/list` e permettere la selezione di un profilo attivo identificato da `avatar_id`. Il requisito ÃfÂ¨ soddisfatto se la lista include almeno un modello locale di fallback (definito in `LOCAL_MODELS`) e, quando il backend ÃfÂ¨ raggiungibile, include anche avatar importati con un URL caricabile del modello `.glb`.

RF2 (Import e caching asset `.glb`): quando ÃfÂ¨ disponibile un URL di export (ad es. da Avaturn), il client deve richiedere l'import con `/avatars/import` e ottenere un URL di cache server-side da usare per il download e l'istanza in scena. Il requisito ÃfÂ¨ soddisfatto se import ripetuti dello stesso URL non creano duplicati, mantengono lo stesso `avatar_id` con URL di cache stabile, e se l'avatar resta caricabile anche dopo riavvio dei servizi.

RF3 (Setup voce persistente per avatar): il client deve consentire la registrazione di un campione vocale e inviarlo al TTS con `/set_avatar_voice`, asso-

ciandolo all'avatar attivo; a completamento, il sistema deve poter generare frasi di attesa tramite `/generate_wait_phrases`. Il requisito ÃfÂ¨ soddisfatto se la registrazione supera una verifica di similaritÃfÂ testuale con soglia definita e se il riferimento vocale risulta riutilizzabile nelle sessioni successive; la persistenza ÃfÂ¨ verificabile tramite endpoint di stato voce e/o generazione TTS senza reinvio del campione.

RF4 (Memoria per-avatar con ingest e retrieval): il sistema deve salvare note e contenuti da file nella memoria dell'avatar tramite `/remember` e `/ingest_file`, e deve usare tale memoria nella generazione contestuale via `/chat`. L'endpoint `/recall` deve essere disponibile per verifiche e diagnostica del contenuto indicizzato. Il requisito ÃfÂ¨ soddisfatto se la memoria ÃfÂ¨ persistente tra sessioni (finchÃfÂ© la directory dati lato server non viene azzerata o cancellata) e se due avatar distinti non condividono documenti indicizzati; l'isolamento puÃfÂ² essere verificato confrontando i risultati di `/recall` a paritÃfÂ di query.

RF5 (Turno vocale end-to-end in push-to-talk): il client deve gestire una conversazione a turni acquisendo audio, inviandolo allo STT con `/transcribe`, inoltrando la trascrizione al RAG con `/chat` e riproducendo la risposta audio tramite streaming con `/tts_stream`. Il requisito ÃfÂ¨ soddisfatto se, per ogni turno, il sistema produce testo trascritto, testo di risposta e avvio del playback audio con transizioni UI coerenti tra ascolto, elaborazione e riproduzione.

RF6 (Ingest multimodale per memoria per-avatar): il sistema deve consentire l'ingestione multimodale nella memoria dell'avatar, includendo descrizione immagine con `/describe_image` e ingestione documentale con `/ingest_file` (con OCR/estrazione testo per PDF e immagini). Il requisito ÃfÂ¨ soddisfatto se, dopo l'operazione, i contenuti risultano recuperabili tramite `/recall` e/o se `/avatar_stats` riporta `has_memory=true` per l'avatar.

3.1.2 Requisiti non funzionali

I requisiti non funzionali sono formulati secondo il modello ISO/IEC 25010 e si concentrano sulle caratteristiche che influenzano la plausibilitÃfÂ dell'interazione vocale e la sostenibilitÃfÂ tecnica del prototipo. In VR la specifica dei requisiti richiede spesso di dettagliare flussi di scena, artefatti e comportamenti; inoltre cambiamenti minimi possono amplificare costi e complessitÃfÂ ,

rendendo utile fissare vincoli di qualitÃfÂ verificabili giÃfÂ in fase architetturale [**Karre2024VReqST**].

RNF1 (Efficienza prestazionale - time behaviour): la latenza percepita deve restare controllata misurando il tempo tra rilascio del comando push-to-talk e primo campione audio riprodotto dal TTS. Per il prototipo si adotta come soglia prestazionale una risposta udibile entro 5 s in esecuzione locale e entro 8 s in deploy pubblico, tracciando timestamp lato client e tempi di risposta dei servizi; lo streaming riduce l'attesa iniziale. Le soglie 5 s/8 s sono trattate come target empirici per limitare il silenzio percepito e distinguere il comportamento locale da quello in deploy. La conformitÃfÂ al requisito ÃfÂ¨ verificata sui log dei turni vocali (timestamp), calcolando percentuali di turni entro soglia e percentili di latenza nei due contesti (locale/pubblico). Approcci a micro-servizi per agenti sociali real-time trattano la latenza come vincolo primario e riportano tempi dell'ordine di pochi secondi fino alla prima emissione vocale in pipeline ottimizzate [**Lin2024Estuary**].

RNF2 (AffidabilitÃfÂ - availability e fault tolerance): durante una sessione di 60 minuti il sistema deve completare con successo tra il 95% e il 97% dei turni vocali, dove un turno ÃfÂ¨ riuscito se STT, `/chat` e `/tts_stream` restituiscono esito valido entro timeout. Nel prototipo sono implementati timeout, retry limitati a `retryCount` (configurabile) e gestione esplicita degli errori lato client; in caso di fallimento il flusso degrada su messaggi di stato UI senza blocchi permanenti. La conformitÃfÂ al requisito ÃfÂ¨ determinata conteggiando turni riusciti e falliti nei log di sessione.

RNF3 (ManutenibilitÃfÂ - modularity e portabilitÃfÂ): la sostituzione di un micro-servizio deve richiedere solo una variazione di configurazione centralizzata lato client (base URL, timeout e policy di retry), mantenendo invariati endpoint e payload attesi. Il requisito si considera soddisfatto quando la stessa build Unity WebGL funziona sia in locale (URL assoluti su loopback) sia in deploy dietro reverse proxy con path `/api/<servizio>`, e quando dopo un aggiornamento o un redeploy viene rieseguita una checklist minima di compatibilitÃfÂ (`/health` e una chiamata funzionale per ciascun servizio).

3.2 Architettura di riferimento di SOULFRAME

3.2.1 Vista d'insieme dei componenti frontend/backend

SOULFRAME adotta un'architettura client–server a componenti, progettata per mantenere sul browser le responsabilitÃfÂ sensibili al frame-rate (interfaccia e resa 3D) e delegare al backend i compiti di inferenza e persistenza. Il client ÃfÂ¨ una build Unity WebGL eseguita nel browser: gestisce la UI, le transizioni di stato dell'esperienza e l'interazione push-to-talk, oltre al rendering e alla visualizzazione dell'avatar. La comunicazione verso il backend avviene via HTTP e viene normalizzata tramite una configurazione centralizzata che astrae la differenza tra esecuzione locale (URL e porte esplicite su loopback) e deploy pubblico (path relativi sotto un unico origin).

Sul backend, la pipeline ÃfÂ¨ scomposta in quattro micro-servizi FastAPI indipendenti, ciascuno con un contratto API mirato. Il servizio Speech-to-Text (STT) basato su Whisper espone l'endpoint `/transcribe` sulla porta 8001 e riceve un file audio per restituire la trascrizione testuale del turno. Il servizio RAG (Retrieval-Augmented Generation) sulla porta 8002 fornisce sia l'orchestrazione conversazionale tramite `/chat` sia le operazioni di memoria per-avatar (`/remember`, `/recall`) e ingestione di contenuti (`/ingest_file`). In questa architettura il RAG funge da punto di convergenza: oltre a costruire il contesto con retrieval, delega a un servizio di supporto esterno le operazioni di Large Language Model (LLM) e di embedding. Il servizio Text-to-Speech (TTS) basato su Coqui XTTS v2, in ascolto sulla porta 8004, espone sia sintesi completa (`/tts`) sia sintesi in streaming (`/tts_stream`); inoltre include endpoint dedicati alle frasi di attesa (`/wait_phrase`, `/generate_wait_phrases`), utili per mascherare tempi morti durante l'elaborazione. Infine, l'Avatar Asset Server sulla porta 8003 gestisce la lista e l'import degli avatar (`/avatars/list`, `/avatars/import`) e serve i modelli `.glb` tramite `/avatars/id/model.glb`, implementando caching server-side degli asset e rendendo persistenti i modelli tra sessioni.

Il servizio di supporto Ollama opera come runtime separato per LLM ed embeddings sulla porta 11434. Nel prototipo il profilo del modello di chat puÃfÂ² variare tra ambiente locale e server con GPU dedicata (profilo lightweight vs profilo standard), mentre il modello embedding resta dedicato al retrieval. Que-

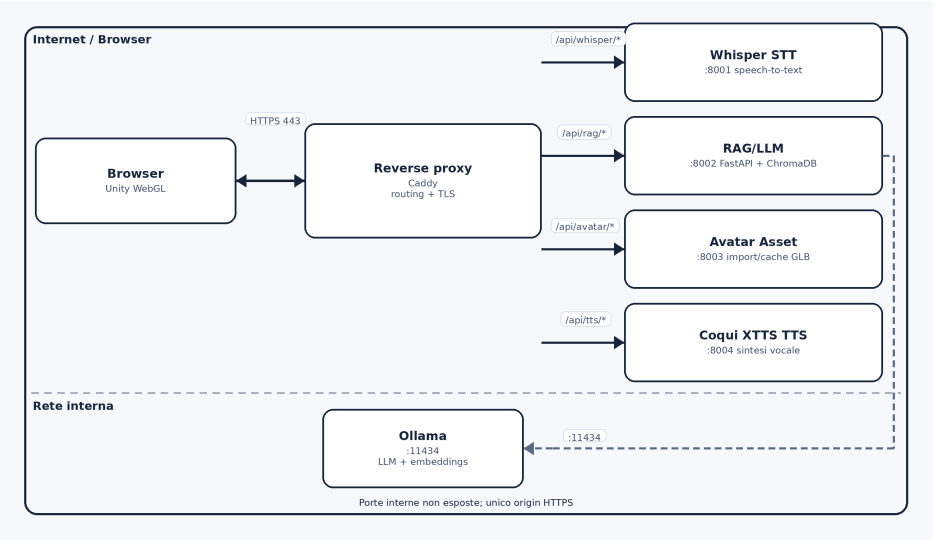

Fig. 3.1: Vista d'insieme dell'architettura a componenti di SOULFRAME: il client Unity WebGL comunica con i micro-servizi backend attraverso il reverse proxy Caddy.

sta configurazione mantiene il micro-servizio RAG focalizzato sull'orchestrazione applicativa (memoria, retrieval e composizione della richiesta) e consente di sostituire o aggiornare i modelli senza modificare il client, a patto di preservare gli endpoint e i formati attesi dal servizio RAG. I dettagli di confronto tra configurazioni modello sono discussi nel Capitolo 4.

In deploy pubblico, l'elemento chiave di integrazione ÃfÂ¨ il reverse proxy Caddy, che svolge due funzioni: terminazione TLS e punto d'ingresso unico per il browser. Il client WebGL comunica quindi con un solo origin HTTPS e invoca le API tramite path riscritti (`/api/whisper/*`, `/api/rag/*`, `/api/avatar/*`, `/api/tts/*`); Caddy inoltra tali richieste alle porte interne 8001–8004, mantenute non esposte verso l'esterno. Questa configurazione riduce la complessitÃfÂ lato browser (stessa origine, routing consistente) e isola i servizi, che restano indirizzabili e gestibili separatamente a livello di processo. Nel prototipo, gli endpoint non implementano autenticazione/autorizzazione applicativa e l'architettura assume una rete di esecuzione controllata. Figura 3.1 riassume questa vista a blocchi, chiarendo sia la separazione di responsabilitÃfÂ tra frontend e backend sia il ruolo del proxy come snodo di comunicazione e sicurezza.

Dal punto di vista operativo, la stessa scomposizione in componenti viene

mantenuta sia in locale sia su server. In ambiente Windows lo script di avvio coordina l'esecuzione dei servizi e applica controlli di base (ad esempio disponibilitÃfÂ delle porte) per ridurre conflitti tra processi durante lo sviluppo. In ambiente Ubuntu, ciascun micro-servizio ÃfÂ¨ gestito come unitÃfÂ `systemd`, con comandi amministrativi che permettono start/stop/status e aggiornamenti senza dover riconfigurare manualmente l'intero stack. Questa impostazione rende esplicito il confine tra runtime del browser (UI e scena) e runtime dei servizi (inferenza e storage), mantenendo l'architettura logica stabile mentre cambiano contesto di esecuzione e modalitÃfÂ di deploy.

3.2.2 Flusso end-to-end audio → testo → risposta → audio

Il flusso conversazionale end-to-end di SOULFRAME ÃfÂ¨ organizzato come una pipeline a turni che parte dall'input vocale dell'utente e termina con la riproduzione della risposta sintetizzata, mantenendo sul client Unity WebGL le responsabilitÃfÂ di interazione e rendering e delegando al backend le fasi di inferenza. L'interazione ÃfÂ¨ di tipo push-to-talk: l'utente tiene premuto `SPACE` per parlare e, al rilascio, il client chiude la registrazione e prepara un file audio in formato WAV, includendo un parametro di lingua nella richiesta. In ambiente WebGL l'acquisizione del microfono e la gestione del contesto audio del browser richiedono un bridge JavaScript, qui realizzato tramite un plugin (`AudioCapture.jslib`) richiamato dal componente di registrazione in Unity, cosÃfÂ¬ da appoggiarsi alle primitive audio disponibili nel runtime WebGL.[1]

Una volta completata la cattura, il client invia l'audio al servizio Speech-to-Text (STT) basato su Whisper tramite una richiesta `POST /transcribe`. Il micro-servizio esegue la trascrizione e restituisce un JSON che contiene il testo riconosciuto nel campo `"text"`. Dal punto di vista architetturale, questa scelta isola l'Automatic Speech Recognition (ASR) in un componente dedicato, rendendo la prima trasformazione del flusso un passaggio esplicito da segnale audio (WAV) a contenuto testuale (stringa), su cui ÃfÂ¨ poi possibile applicare logiche conversazionali e di memoria.

[1] Unity Technologies, *Audio in WebGL*, https://docs.unity3d.com/6000.2/Documentation/Manual/webgl-audio.html, Accessed: 2026-02-22.

La trascrizione viene quindi inoltrata al servizio RAG/LLM con una richiesta `POST /chat` che include l'identificativo dell'avatar (`avatar_id`). In questo stadio il backend svolge la funzione di orchestratore: recupera contesto dalla memoria per-avatar, costruita su un database vettoriale persistente (ChromaDB) e popolata tramite embedding generati con un modello dedicato (nella configurazione corrente `nomic-embed-text` erogato da Ollama). Il retrieval, quando disponibile memoria, produce un insieme di passaggi contestuali che vengono integrati nel prompt e usati per vincolare e arricchire la generazione. Il servizio invoca poi il modello linguistico per la risposta (nella configurazione corrente `llama3:8b-instruct-q4_K_M` via Ollama), ottenendo un testo finale che viene restituito al client come contenuto della risposta. In questa fase si concentrano sia la dipendenza dalla memoria dell'avatar sia la variabilitÃfÂ computazionale dovuta alla generazione, motivo per cui la gestione della latenza percepita diventa parte integrante del disegno architetturale.

Ricevuto il testo di risposta, il client attiva la sintesi vocale tramite il servizio Text-to-Speech (TTS) basato su Coqui XTTS v2. La richiesta avviene preferibilmente tramite l'endpoint di streaming `/tts_stream`, includendo `avatar_id` e lingua: il servizio recupera il profilo vocale associato all'avatar (voice cloning) e produce audio progressivamente, consentendo al client di iniziare il playback non appena arrivano i primi byte dello stream. Lo streaming riduce il tempo tra fine turno utente e inizio della risposta udibile, perchÃfÂ© sposta l'attesa dal completamento dell'intero file audio alla sola generazione dell'incipit, migliorando la continuitÃfÂ percepita anche quando la risposta ÃfÂ¨ lunga.

SOULFRAME introduce inoltre strategie specifiche per mitigare i tempi morti piÃfÂ¹ evidenti. La prima ÃfÂ¨ un warmup del TTS durante il boot: il servizio, all'avvio, genera una frase breve (ad esempio "Ciao.") per inizializzare modello e dipendenze, riducendo il costo della prima richiesta reale; in parallelo, il frontend mostra un pannello di caricamento dedicato e rende disponibili le modalitÃfÂ operative solo quando il TTS risulta pronto. La seconda strategia riguarda le wait phrases: il sistema puÃfÂ² riprodurre brevi vocalizzi o frasi di attesa pre-generate (ad esempio "hm" o "un secondo") mentre la pipeline STT→RAG/LLM→TTS ÃfÂ¨ in corso, cosÃfÂ¬ da evitare silenzi prolungati e dare un segnale immediato di reattivitÃfÂ . Queste frasi sono associate al profilo

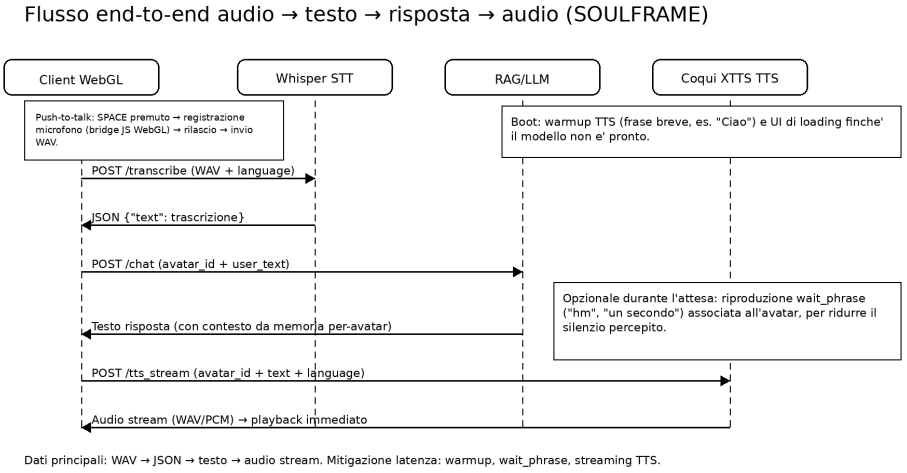

Fig. 3.2: Flusso end-to-end di una richiesta conversazionale in SOULFRAME: dall'acquisizione audio push-to-talk alla riproduzione della risposta vocale.

vocale dell'avatar e vengono generate una tantum o rigenerate quando necessario; dal punto di vista architetturale sono un canale asincrono di feedback che non altera il contenuto della risposta, ma agisce sulla percezione temporale dell'utente.

Figura 3.2 sintetizza la sequenza temporale e i formati dati scambiati tra componenti, mettendo in evidenza dove avvengono le trasformazioni principali (WAV→testo, testo→testo contestualizzato, testo→audio) e dove si innestano le tecniche di riduzione della latenza percepita (warmup, wait phrases, streaming). La stessa vista ÃfÂ¨ utile anche per chiarire che, in deploy con reverse proxy, gli endpoint REST restano invariati a livello logico e vengono solo raggiunti tramite path riscritti sotto un unico origin HTTPS.

3.2.3 Flusso di gestione avatar (creazione, import, cache, rendering)

Il flusso di gestione degli avatar in SOULFRAME affianca due percorsi distinti, che convergono poi sulla stessa esperienza di utilizzo in scena: da un lato gli avatar locali pre-inclusi nella build, dall'altro gli avatar importati, creati dall'utente tramite Avaturn. L'obiettivo architetturale ÃfÂ¨ garantire che il client Unity WebGL possa sempre offrire una libreria minima di modelli selezionabili, mantenendo allo stesso tempo un canale controllato per acquisire, validare e ser-

vire asset esterni in formato `.glb` senza dipendere direttamente da URL remoti durante il rendering.

Gli avatar locali costituiscono il percorso piÃfÂ¹ semplice: i modelli `LOCAL_model1` e `LOCAL_model2` sono sempre presenti e vengono esposti nella lista restituita dal backend insieme agli avatar importati. A livello di interazione, l'utente naviga la libreria tramite l'interfaccia a carosello e seleziona un profilo; il client aggiorna quindi lo stato dell'avatar corrente e avvia il caricamento del modello nella scena. In questa modalitÃfÂ non ÃfÂ¨ necessario alcun passaggio di import o caching, poichÃfÂ© l'asset ÃfÂ¨ giÃfÂ disponibile nel pacchetto o in una posizione nota al runtime Unity, e il flusso si riduce a selezione e rendering.

Il percorso degli avatar importati introduce invece un passaggio di creazione esterna, gestito come overlay nel browser. Dal menu del client, l'utente avvia la creazione aprendo l'esperienza Avaturn in un iframe: `AvaturnWebController.cs` richiede l'apertura dell'overlay e delega la gestione dell'iframe al bridge JavaScript, che incapsula il ciclo di vita dell'interazione (apertura, caricamento, chiusura) e la ricezione dell'evento di export. Quando l'utente termina la personalizzazione, Avaturn produce un export del modello `.glb` e il bridge (`AvaturnBridge.jslib`) inoltra al runtime Unity un payload JSON con URL, `avatarId` e metadati (ad esempio `gender`, `bodyId`, `urlType`) sfruttando il meccanismo standard di interoperabilitÃfÂ Unity WebGL tra JavaScript e C# basato su plugin `.jslib` e chiamate di messaggistica verso GameObject.[2] Il client riceve il JSON e lo delega al gestore avatar, che avvia la fase di import.

L'import nel backend avviene tramite l'endpoint `/avatars/import` del servizio `avatar_asset_server.py`. La richiesta include `avatar_id`, URL del file `.glb` esportato e metadati utili a ricostruire il profilo dell'avatar. Il server scarica il file e lo memorizza in una directory di storage dedicata (`avatar_store/`), aggiornando un file di metadati JSON che mantiene la lista degli avatar importati. Questo passaggio realizza una cache server-side stabile: il client non deve piÃfÂ¹ dipendere dall'URL originario di Avaturn durante le sessioni successive, ma puÃfÂ² caricare il modello da un endpoint controllato e coerente con il deploy. Il modello viene poi esposto tramite `GET /avatars/{id}/model.glb` e pubblicato al client

[2] Doc ufficiale: Unity WebGL – Interacting with browser scripting, https://docs.unity3d.com/6000.2/Documentation/Manual/webgl-interacting-browser-js.html.

come `cached_glb_url` all'interno della risposta di `/avatars/list`, che aggrega sempre avatar locali e importati in un'unica libreria.

Un aspetto rilevante di robustezza ÃfÂ¨ la logica di self-healing dei metadati implementata dal servizio asset: in fase di listing, il server risolve `file_path` verificando l'esistenza del file e, se necessario, ricostruisce il collegamento individuando il modello corrispondente nella cache (ad esempio tramite pattern sul nome e timestamp). In presenza di deploy o migrazioni che cambiano percorsi assoluti o struttura delle directory, questa strategia riduce la probabilitÃfÂ di riferimenti obsoleti e mantiene la lista coerente con lo stato effettivo dello storage.

Dal lato client, una volta ottenuto `cached_glb_url`, il runtime Unity scarica il modello e lo istanzia nello spawn point dell'avatar, aggiornando i riferimenti di scena e lo stato dell'avatar corrente. Il caricamento e lo switching vengono orchestrati dal gestore avatar e dall'interfaccia della libreria, cosÃfÂ¬ che la transizione tra profili sia percepita come un'operazione di selezione indipendente dalla provenienza (locale o importata). Figura 3.3 riassume i due percorsi e il punto di convergenza, chiarendo dove intervengono iframe Avaturn, bridge WebGL, caching server-side e download del `.glb` prima del rendering in Unity.

Dopo la selezione o creazione, il flusso passa al setup voce dell'avatar. In questa fase la soglia del 70% non rappresenta una verifica biometrica: il client registra un campione, lo invia a `/transcribe` e confronta la trascrizione con la frase attesa mostrata in UI, applicando una metrica di similaritÃfÂ testuale. Il campione viene accettato solo se lo score supera la soglia `0.7`; in caso contrario l'utente viene invitato a ripetere la registrazione. Superata la verifica, il client persiste il profilo vocale tramite `/set_avatar_voice` e avvia la generazione delle frasi di attesa con `/generate_wait_phrases`.

Completato il setup voce, l'onboarding verifica la memoria per-avatar tramite `/avatar_stats?avatar_id=...` (campo `has_memory`). Se la memoria ÃfÂ¨ assente, il client indirizza a `SetupMemory`, che offre tre modalitÃfÂ : nota manuale (`/remember`, con metadati di provenienza, ad es. `source_type=manual`), ingestione file `.pdf`/`.txt` via `/ingest_file` (estrazione del contenuto e indicizzazione nella memoria vettoriale), e descrizione immagine via `/describe_image` con salvataggio della descrizione in memoria. Questo passaggio popola dati testuali e metadati associati all'`avatar_id`, rendendo il retrieval disponibile giÃfÂ dai

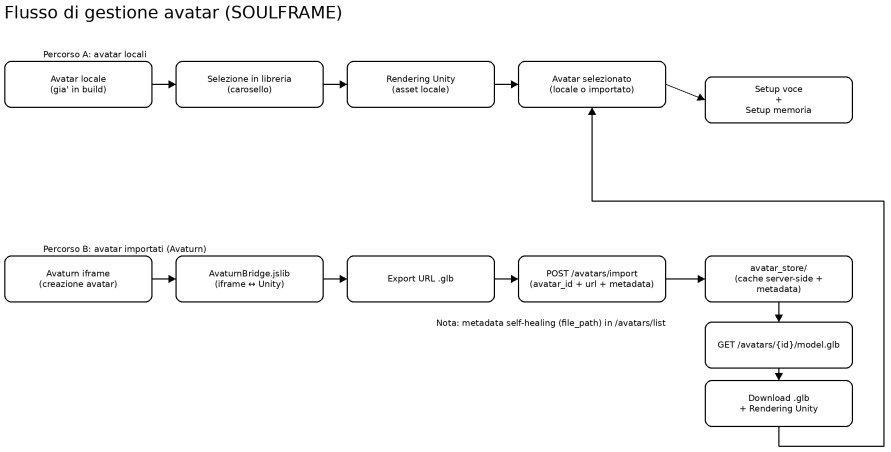

Fig. 3.3: Flusso di gestione avatar in SOULFRAME: dalla creazione tramite Avaturn alla cache server-side e al rendering nel client Unity.

primi turni di `/chat`; la presenza dei contenuti ÃfÂ¨ verificabile sia con `/recall` sia con `/avatar_stats`. In termini di stato applicativo, il client puÃfÂ² applicare un redirect automatico al setup mancante prima dell'accesso a `MainMode`. Completato setup avatar (voce/memoria), l'avatar diventa soggetto attivo della pipeline conversazionale descritta in 3.2.2.

3.3 Componenti implementati

Questa sezione descrive i principali componenti effettivamente realizzati nel prototipo SOULFRAME, mettendo in relazione le scelte implementative con l'architettura logica presentata nella Sezione 3.2. L'obiettivo e' chiarire quali moduli concretizzano i flussi discussi nel capitolo, senza entrare nei dettagli piu' minuti di singole funzioni o ottimizzazioni, che verranno affrontati nel Capitolo 4. In particolare, si analizzano l'integrazione di Avaturn nel client Unity WebGL, l'implementazione dei micro-servizi AI e la gestione della persistenza per profilo avatar.

3.3.1 Integrazione Avaturn nel frontend Unity WebGL

L'integrazione di Avaturn nel frontend ha l'obiettivo di consentire la creazione di avatar personalizzati senza uscire dall'applicazione, mantenendo l'esperienza

coerente con un prototipo che gira interamente nel browser. In SOULFRAME questo risultato viene ottenuto incapsulando l'editor di Avaturn in un overlay basato su iframe, aperto dal client Unity WebGL e gestito tramite un bridge JavaScript dedicato. Il flusso architetturale complessivo (export del `.glb`, import nel backend e caching server-side) e' descritto nella Sezione 3.2; qui si dettaglia invece come il client avvia e governa l'interazione con l'editor web.

Il punto di ingresso lato Unity e' `AvaturnWebController.cs`, un componente che espone in inspector l'URL di Avaturn (campo `avaturnUrl`) e mantiene un riferimento a `UIFlowController`. In fase di avvio, se tale riferimento non e' cablato, il componente tenta un recupero automatico tramite ricerca in scena. Il metodo `OnClick_NewAvatar()` implementa il comportamento di apertura in modo dipendente dalla piattaforma: in editor, per motivi di praticita' e debugging, l'URL viene aperto con `Application.OpenURL` e viene anche aggiornato un messaggio di stato nella UI; in build WebGL, invece, la chiamata viene inoltrata alla funzione JavaScript `OpenAvaturnIframe` tramite `DllImport("__Internal")`, passando tre parametri: l'URL, il nome del `GameObject` che ricevera' i callback e il nome del metodo callback (`"OnAvatarJsonReceived"`). Su piattaforme diverse da WebGL viene emesso un warning esplicito, rendendo evidente che questo percorso e' progettato per il runtime browser.

Il bridge e' implementato in `AvaturnBridge.jslib`, che definisce `OpenAvaturnIframe` e la registra nella libreria WebGL tramite `mergeInto`. All'invocazione, i parametri ricevuti da C# vengono convertiti da puntatori a stringhe e usati per configurare una comunicazione di ritorno verso Unity basata su `SendMessage`. Il bridge incapsula l'apertura dell'overlay creando dinamicamente un elemento `div` a schermo intero, con un header e un pulsante di chiusura, e un container dedicato all'iframe. Contestualmente, vengono inviati eventi di stato a Unity chiamando `OnWebOverlayOpened` e `OnWebOverlayClosed` sul `GameObject` passato dal client. L'interoperabilita' tra C# e JavaScript in WebGL sfrutta il meccanismo standard dei plugin `.jslib` e l'invio di messaggi verso un `GameObject` Unity.[3] Per migliorare l'usabilita' in presenza di input da tastiera, il bridge introduce anche una logica di refocus sul canvas Unity dopo la chiusura dell'overlay, cosi' da ripristinare correttamente il focus del browser al runtime WebGL.

[3] Doc ufficiale: Unity WebGL — Interaction with browser scripting, https://docs.unity3d.com/6000.2/Documentation/Manual/webgl-interacting-browser-js.html.

Il caricamento dell'editor Avaturn non avviene tramite una pagina statica, ma attraverso l'iniezione di uno script `type="module"` che importa e inizializza `AvaturnSDK` all'interno del container. Una volta inizializzato, il bridge sottoscrive l'evento `"export"` dell'SDK: quando l'utente completa la personalizzazione, l'handler costruisce un payload JSON minimale con i campi `url`, `avatarId`, `gender`, `bodyId` e `urlType` e lo inoltra a Unity richiamando il callback specificato (`OnAvatarJsonReceived`) sul `GameObject` registrato. In caso di errore o chiusura manuale, il bridge invia invece un payload con `status` valorizzato (ad esempio `"error"` o `"closed"`), permettendo al frontend di aggiornare lo stato UI anche in assenza di export.

Sul lato C#, `AvaturnWebController.cs` riceve il JSON nel metodo `OnAvatarJsonReceived(string json)` e lo delega a `UIFlowController.OnAvatarJsonReceived`, mantenendo il controller web come adattatore sottile tra bridge e logica applicativa. In `UIFlowController.cs` il JSON viene parsato in una struttura serializzabile per intercettare i casi di chiusura o errore e aggiornare lo stato interno dell'overlay; successivamente, il payload viene inoltrato al gestore avatar, che avvia la fase di import verso il backend. In questo modo, l'integrazione Avaturn rimane confinata a un segmento ben definito del frontend: l'overlay e i callback web gestiscono l'interazione esterna, mentre il resto del client continua a operare sul medesimo modello di stato e sulle stesse transizioni descritte a livello architetturale nella Sezione 3.2.

Infine, `AvaturnSystem.cs` svolge un ruolo di supporto e compatibilita': disabilita gli elementi UI del prefab Avaturn originale presenti in scena e, in ambiente WebGL, instrada l'apertura dell'iframe verso `AvaturnWebController`, rendendo esplicita la scelta di un'integrazione custom basata su overlay e bridge, invece dell'iframe controller di default fornito dal pacchetto Avaturn. Questa separazione consente di mantenere il percorso WebGL controllato e coerente con le esigenze del prototipo, senza vincolare la logica di creazione avatar alla UI del prefab esterno.

3.3.2 Backend AI a micro-servizi (Whisper, RAG, TTS, Avatar Asset)

Il backend di SOULFRAME e' realizzato come insieme di micro-servizi indipendenti basati su FastAPI, avviati con `uvicorn` su porte dedicate (8001–8004).

Questa scelta rende esplicita la separazione delle responsabilita' e consente di verificare disponibilita' e corretto instradamento delle richieste con controlli semplici (`/health`) prima di abilitare il flusso conversazionale. Tabella **??** sintetizza i quattro componenti e i relativi contratti API: il riepilogo e' utile sia in fase di sviluppo (per individuare rapidamente dipendenze e punti di failure) sia per la lettura del capitolo, perche' connette porte, endpoint e responsabilita' operative in un'unica vista.

Il micro-servizio Whisper implementa la componente Speech-to-Text (STT) in modo volutamente minimale, privilegiando un contratto stabile rispetto a ottimizzazioni premature. In `whisper_server.py` il modello viene caricato una sola volta all'avvio tramite `whisper.load_model`, usando la variabile d'ambiente `WHISPER_MODEL` (default `"small"`). L'endpoint `POST /transcribe` riceve un `UploadFile` e un campo form `language` (default `"it"`), salva temporaneamente il contenuto su disco con `tempfile.NamedTemporaryFile` e invoca `model.transcribe` specificando la lingua normalizzata. La risposta viene ridotta al solo campo `"text"`, cosi' da rendere esplicito il passaggio architetturale audio→testo e minimizzare la dipendenza del client da dettagli interni del modello. Il file temporaneo viene sempre rimosso nel blocco `finally`, limitando l'accumulo di artefatti in caso di errori o richieste interrotte.

Il servizio RAG (Retrieval-Augmented Generation) rappresenta il nucleo di orchestrazione conversazionale e di memoria. In `rag_server.py` la configurazione esterna e' centralizzata in variabili d'ambiente, tra cui `OLLAMA_HOST` (default `http://127.0.0.1:11434`), `CHAT_MODEL` (default `llama3:8b-instruct-q4_K_M`) ed `EMBED_MODEL` (default `nomic-embed-text`). L'endpoint `POST /chat` riceve una richiesta strutturata con `avatar_id` e testo utente; se la collezione dell'avatar contiene documenti, il server calcola l'embedding della query tramite Ollama (`/api/embed`), esegue retrieval e compone un prompt contestualizzato prima di chiamare Ollama per la generazione (`/api/chat`). La memoria e' persistente e isolata per avatar: per ogni `avatar_id` viene creato un database ChromaDB su filesystem sotto `rag_store/` (configurabile con `RAG_DIR`), e la collezione viene ottenuta tramite `get_or_create_collection(name="memory")`. L'endpoint `POST /remember` consente di aggiungere note testuali alla memoria, mentre `POST /recall` fornisce una modalita' di interrogazione utile per diagnostica e verifica

dei contenuti indicizzati. L'endpoint `GET /avatar_stats` espone metriche minime (conteggio e flag `has_memory`) usate dal frontend per decidere se guidare l'utente a un setup di memoria.

Dal punto di vista del retrieval, il servizio implementa una ricerca ibrida: prima seleziona candidati con similarita' vettoriale su ChromaDB, poi ricalcola un ranking lessicale sui candidati con BM25 (libreria `rank-bm25`) e combina i punteggi con un peso esplicito (`bm25_weight`). Questo approccio riduce la dipendenza da una singola metrica e consente di recuperare frammenti rilevanti anche quando la query contiene termini discriminanti (nomi propri, parole chiave) che la sola similarita' semantica puo' attenuare. La parte di ingestione e' implementata in `POST /ingest_file`, che accetta un file associato a `avatar_id` e applica una pipeline differenziata per estensione: i PDF vengono processati con estrazione testo basata su PyMuPDF e OCR, le immagini passano tramite OCR con `pytesseract`, mentre i file di testo vengono letti come plain text. Il contenuto viene poi spezzato in chunk (parametri `RAG_CHUNK_CHARS` e `RAG_CHUNK_OVERLAP`), deduplicato a livello di chunk e indicizzato calcolando embedding in batch. Infine, `POST /describe_image` offre una descrizione testuale di immagini tramite un client Gemini (abilitato solo se `GEMINI_API_KEY` e' configurata) e, opzionalmente, permette di salvare la descrizione in memoria con metadati coerenti (`source_type="image_description"`), mantenendo l'output utilizzabile dal retrieval come semplice testo.

Il micro-servizio di sintesi vocale Coqui implementa la componente Text-to-Speech (TTS) e gestisce i profili vocali per avatar. In `coqui_tts_server.py` il modello e la lingua di default sono configurabili (`COQUI_TTS_MODEL`, default `xtts_v2`; `COQUI_LANG`, default `"it"`), cosi' come la directory di persistenza dei profili (`COQUI_AVATAR_VOICES_DIR`). Il profilo vocale viene caricato tramite `POST /set_avatar_voice`, che riceve un `speaker_wav` e lo normalizza/salva come `voices/avatars/<avatar_id>/reference.wav`; `GET /avatar_voice` consente di verificare presenza e dimensione del riferimento, mentre `DELETE /avatar_voice` consente di rimuovere il profilo. La sintesi principale avviene con `POST /tts`, che restituisce audio in streaming (risposta `StreamingResponse`) e include header informativi sul formato e sull'origine del riferimento vocale usato; e' disponibile anche `POST /tts_json` per ottenere lo stesso audio codificato in base64 quando

lo streaming non e' pratico. Per ridurre la latenza percepita e supportare il flusso conversazionale descritto in Sezione 3.2, il servizio implementa frasi di attesa: `POST /generate_wait_phrases` pre-genera e memorizza clip brevi per uno specifico `avatar_id`, mentre `GET /wait_phrase` serve una singola clip selezionata per nome. La fase di warmup e' supportata tramite parametri di avvio (`COQUI_WARMUP_ON_STARTUP`, `COQUI_WARMUP_TEXT`), cosi' da inizializzare modello e dipendenze prima dell'utilizzo interattivo.

Infine, `avatar_asset_server.py` gestisce l'import e la distribuzione degli asset `.glb` degli avatar, includendo sia modelli locali di fallback sia modelli creati tramite Avaturn. `GET /avatars/list` costruisce una lista unificata che include `LOCAL_MODELS` (ad esempio `LOCAL_model1` e `LOCAL_model2`) e gli avatar importati, arricchendo questi ultimi con un `cached_glb_url` calcolato a partire dalla base URL della richiesta. L'import avviene con `POST /avatars/import`: il server valida l'URL, calcola un hash SHA-256 (`url_hash`) e applica deduplicazione cercando un record gia' esistente con lo stesso hash; se presente e il file e' risolvibile, viene restituita una risposta con `dedup=True` senza riscaricare l'asset. In caso contrario, il `.glb` viene scaricato e salvato in `avatar_store/models/` con un nome che incorpora hash e `avatar_id`. La risoluzione del percorso dell'asset e' robusta rispetto a migrazioni o metadati obsoleti grazie a `resolve_avatar_file_path`, che tenta sia il path registrato sia pattern di fallback basati su `avatar_id` o `url_hash`. I metadati sono conservati in `avatar_store/avatars.json` e aggiornati con scrittura atomica tramite file temporaneo `.tmp` e `replace`, riducendo il rischio di corruzione in caso di arresti improvvisi. Il download del modello avviene infine tramite `GET /avatars/avatar_id/model.glb` (risposta `FileResponse`), mentre `DELETE /avatars/avatar_id` consente la rimozione controllata di un avatar importato insieme al suo file in cache.

3.3.3 Persistenza e gestione dati per avatar

La persistenza dei dati in SOULFRAME e' organizzata su filesystem e segue un criterio di separazione per `avatar_id`: ogni profilo mantiene in modo indipendente asset 3D, profilo vocale e memoria conversazionale. Questa scelta evita l'introduzione di un DBMS relazionale centrale nel prototipo e rende esplicito il legame tra identita' dell'avatar e risorse necessarie a renderlo operativo, coeren-

temente con i requisiti di cache stabile, voce riutilizzabile e memoria persistente discussi in Sezione 3.1. La ricostruzione dello stato dopo un riavvio avviene rileggendo le directory di persistenza e, quando necessario, applicando meccanismi di riparazione automatica dei metadati.

Per gli asset avatar importati, il servizio `avatar_asset_server.py` definisce uno store dedicato `avatar_store/`, con i modelli `.glb` salvati in `avatar_store/models/` e un file metadati `avatar_store/avatars.json` che descrive gli avatar disponibili. Lo store viene inizializzato in modo idempotente: se necessario vengono create le directory e, se il file JSON non esiste o risulta illeggibile, viene ripristinata una struttura minima vuota. La scrittura dei metadati adotta un approccio atomico, producendo prima un file temporaneo `avatars.json.tmp` e poi sostituendo il file definitivo, riducendo il rischio di corruzione in caso di interruzioni. Anche il download dei modelli e' protetto da una strategia simile: il `.glb` viene salvato come `.part` e rinominato solo a completamento. Per limitare duplicazioni, l'import applica una deduplicazione basata su hash SHA-256 dell'URL sorgente (`url_hash`) e, se un avatar gia' presente e' ancora recuperabile su disco, restituisce direttamente l'URL cache (`cached_glb_url`) senza riscaricare il file. In aggiunta, la funzione di risoluzione del percorso implementa self-healing: se `file_path` non punta piu' a un file valido, il server tenta di ricostruire il collegamento cercando in `models/` un candidato compatibile prima per `avatar_id` e poi per `url_hash`, aggiornando `avatars.json` quando individua un percorso corretto.

Il profilo vocale dell'avatar e le relative risorse di attesa sono persistite dal servizio Text-to-Speech (TTS) `coqui_tts_server.py` nella directory `voices/avatars/` (configurabile tramite `COQUI_AVATAR_VOICES_DIR`). Il server normalizza l'identificativo con una funzione di sanitizzazione e salva il campione di riferimento come `voices/avatars/<avatar_id>/reference.wav`. La scelta di un percorso fisso e per-avatar consente al sistema di riutilizzare la stessa voce tra sessioni senza richiedere ogni volta il reinvio dell'audio, mentre l'endpoint `/avatar_voice` espone un controllo semplice e verificabile sull'esistenza e sulla dimensione del file salvato. Nella stessa directory vengono persistite anche le wait phrases come `wait_<chiave>.wav`, generate esplicitamente tramite `/generate_wait_phrases`. Il servizio supporta inoltre una generazione lazy: se

una specifica wait phrase manca, `/wait_phrase` puo' rigenerarla on-demand a partire dal riferimento vocale disponibile e salvarla su disco, cosi' da mantenere la conversazione robusta anche quando file secondari non sono presenti o sono stati rimossi.

La memoria conversazionale e i contenuti indicizzati sono gestiti dal servizio Retrieval-Augmented Generation (RAG) `rag_server.py` attraverso ChromaDB, persistendo i dati in `rag_store/` (configurabile tramite `RAG_DIR`). Il server normalizza `avatar_id` in una chiave sicura e crea una sottodirectory dedicata `rag_store/<avatar_key>/` che funge da database persistente dell'avatar; all'interno di tale database viene mantenuta una singola collezione `memory`. Questa impostazione realizza l'isolamento per-avatar a livello di storage: due avatar distinti non condividono collezioni e non possono recuperare contenuti dell'altro, a meno di un'alterazione manuale del filesystem. L'endpoint `/avatar_stats` fornisce una misura diretta della presenza di memoria tramite conteggio dei documenti (`count`) e un booleano `has_memory`; per la gestione operativa e il reset controllato e' disponibile anche `/clear_avatar`, che puo' eliminare la collezione e, in modalita' hard, rimuovere la directory persistente su disco.

Nel complesso, un avatar puo' essere considerato completo quando coesistono tre elementi: un modello `.glb` raggiungibile tramite il servizio asset (verificabile tramite `/avatars/list` e `/avatars/id/model.glb`), un riferimento vocale presente in `voices/avatars/<avatar_id>/reference.wav` (verificabile con `/avatar_voice`) e una memoria RAG non vuota (verificabile con `/avatar_stats`). La separazione in store specializzati permette di aggiornare o ripristinare un singolo aspetto dell'avatar (asset, voce o memoria) senza invalidare gli altri, e rende trasparente il legame tra persistenza e requisiti funzionali di caching, riuso della voce e continuita' del contesto tra sessioni, come richiamato in Sezione 3.1.

3.4 Setup e deploy operativo

Il setup e il deploy operativo rendono concreta l'architettura descritta nella Sezione 3.2, traducendo la scomposizione in componenti in procedure ripetibili di avvio, arresto e verifica. SOULFRAME mantiene la stessa struttura logica in locale e in produzione, ma cambia l'orchestrazione dei processi e il punto di

ingresso di rete, in coerenza con i vincoli di portabilita' e manutenibilita' discussi in Sezione 3.1.

3.4.1 Ambiente locale Windows

In ambiente Windows lo stack e' progettato per lo sviluppo e i test rapidi: i servizi backend girano su `127.0.0.1` e la build Unity WebGL viene servita in HTTP su `http://localhost:8000`. Il provisioning iniziale e' automatizzato da `setup_soulframe_windows.bat`, che verifica la presenza della cartella `SOULFRAME_AI/backend`, crea un virtual environment (`venv`) nella posizione di default `backend/venv` (o in un percorso fornito dall'utente), aggiorna `pip` e installa le dipendenze da `requirements.txt`. Lo script consente inoltre di salvare una chiave `GEMINI_API_KEY` in `backend/gemini_key.txt`, rendendo ripetibile l'abilitazione delle funzionalita' opzionali di descrizione immagine lato RAG senza dover riconfigurare manualmente ogni sessione.

L'avvio e la gestione dei processi sono centralizzati in `ai_services.cmd`, che puo' essere invocato sia in modalita' non interattiva (`ai_services.cmd 1/2/3` per start/stop/restart) sia tramite menu testuale. Lo script definisce le porte dei componenti (`8001` Whisper STT, `8002` RAG, `8003` Avatar Asset, `8004` Coqui TTS, `11434` Ollama e `8000` build server) e imposta variabili d'ambiente che parametrizzano i modelli e il comportamento di inferenza, tra cui `WHISPER_MODEL`, `CHAT_MODEL`, `EMBED_MODEL` e un set di parametri per la generazione (`CHAT_TEMPERATURE`, `CHAT_TOP_P`, `CHAT_REPEAT_PENALTY`). Per la sintesi, lo script preconfigura il modello XTTS v2 e i path di persistenza voce tramite `COQUI_TTS_MODEL`, `COQUI_AVATAR_VOICES_DIR` e la scelta del dispositivo (`COQUI_TTS_DEVICE=cuda`), in modo che il processo erediti automaticamente tali impostazioni all'avvio.

La procedura di start e' sequenziale e include controlli di disponibilita' delle porte prima di creare nuovi processi. `ai_services.cmd` usa `netstat` per verificare se una porta e' gia' in ascolto e, in caso contrario, avvia ogni micro-servizio in una finestra separata minimizzata tramite `start ... uvicorn <server>:app -host 127.0.0.1 -port <porta>`. Ollama viene avviato solo se non risulta gia' attivo, richiamando `ollama serve`. A conclusione, viene avviato un server statico per la build WebGL nella directory `Build` tramite `python -m http.server 8000` e viene

aperto il browser all'URL locale. Questo assetto riduce il tempo tra modifica del codice e verifica end-to-end, mantenendo una separazione netta tra interfaccia e servizi.

La procedura di stop e' impostata per ridurre errori accidentali: lo script termina i processi Python legati alle porte dei servizi, ma prima verifica che il PID corrisponda effettivamente a `python.exe` (evitando di chiudere applicazioni non correlate). La chiusura di Ollama e' gestita separatamente con una terminazione forzata del processo in ascolto sulla porta `11434`. In questo modo l'ambiente locale puo' essere riportato rapidamente a uno stato pulito senza interventi manuali su task manager o porte residue.

3.4.2 Ambiente server Ubuntu

In ambiente Ubuntu il deploy e' orientato all'esposizione pubblica del prototipo mantenendo i micro-servizi su rete interna e offrendo al browser un unico origin HTTPS. Il provisioning e' automatizzato da `setup_soulframe_ubuntu.sh`, eseguito come root, che organizza l'installazione sotto `/opt/soulframe`: i sorgenti backend risiedono in `/opt/soulframe/backend`, la build WebGL viene collocata in `/opt/soulframe/webgl`, e l'ambiente Python viene creato in `/opt/soulframe/.venv`. La configurazione runtime viene centralizzata in `/etc/soulframe/soulframe.env`, mentre parametri legati alle politiche di spegnimento per inattivita' sono separati in `/etc/soulframe/idle.env`. Lo script imposta inoltre valori di default coerenti con l'uso server, tra cui `CHAT_MODEL_DEFAULT=llama3.1:8b`, `EMBED_MODEL_DEFAULT=nomic-embed-text` e `WHISPER_MODEL_DEFAULT=medium`, lasciando la possibilita' di override tramite variabili d'ambiente.

L'orchestrazione dei servizi avviene tramite unit `systemd` generate dal setup: per ciascun micro-servizio viene creato un file `.service` (`soulframe-whisper.service`, `soulframe-rag.service`, `soulframe-avatar.service`, `soulframe-tts.service`) che definisce `WorkingDirectory=/opt/soulframe/backend`, carica la configurazione da `EnvironmentFile=/etc/soulframe/soulframe.env` e avvia `uvicorn` sul loopback (`-host 127.0.0.1`) e sulla porta prevista. La gestione include politiche di riavvio (`Restart=always`, `RestartSec=5`) e timeout di startup differenziati, in modo che un riavvio del server ripristini automaticamente lo stack senza interven-

to manuale.[4] Per Ollama viene creato un wrapper `soulframe-ollama.service` che delega a `systemctl start/stop ollama`, e un target `soulframe.target` aggrega i servizi per facilitare start/stop di gruppo.

L'esposizione verso l'esterno e' demandata a Caddy, configurato dal setup tramite un `Caddyfile` che serve i file statici della build WebGL da `/opt/soulframe/webgl` e gestisce la terminazione Transport Layer Security (TLS) e il reverse proxy delle API. Le route sono mappate tramite `handle_path` su `/api/whisper/*`, `/api/rag/*`, `/api/avatar/*` e `/api/tts/*`, inoltrando le richieste alle porte interne `8001-8004`; e' presente anche una regola di compatibilita' `/avatars/*` che proxyfiera verso il servizio asset per gestire eventuali URL legacy restituiti in precedenza.[5] In questo assetto il browser non accede mai direttamente alle porte dei micro-servizi, ma solo al dominio configurato, mentre la stessa build Unity WebGL resta riutilizzabile grazie alla configurazione centralizzata degli endpoint lato client (come discusso in Sezione 3.1).

3.4.3 Servizi di supporto (systemd, Caddy, script amministrativi)

Oltre ai micro-servizi, il deploy include strumenti di supporto pensati per ridurre l'attrito operativo e rendere ripetibili manutenzione e troubleshooting. `systemd` fornisce gestione del ciclo di vita (avvio al boot, restart su failure, raccolta log via `journalctl`) e, tramite `soulframe.target`, consente di trattare lo stack come un'unica unita' logica senza perdere la possibilita' di intervenire su un singolo servizio. Il setup installa anche un comando rapido `sfctl` in `/usr/local/bin/sfctl`, che espone un'interfaccia unificata `start|stop|restart|status|logs` sia per un servizio specifico (whisper/rag/avatar/tts/ollama) sia per l'intero gruppo, rendendo piu' immediata la diagnosi durante una sessione di debug server-side.

Caddy svolge un ruolo complementare: oltre a offrire un origin HTTPS unico per il browser, uniforma il routing delle API e centralizza l'osservabilita' tramite access log su `/var/log/caddy/access.log`. Questo log non viene usato solo per audit o debug, ma alimenta anche una politica di gestione risorse: il setup

[4] Doc ufficiale: systemd.service — Service unit configuration, https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html.

[5] Doc ufficiale: Caddy — reverse_proxy directive, https://caddyserver.com/docs/caddyfile/directives/reverse_proxy.

installa `idle_shutdown.sh` e configura `soulframe-idle-shutdown.service` con un timer `soulframe-idle-shutdown.timer` che esegue un controllo periodico (ogni 60 secondi) dell'attivita' recente, basandosi sulle richieste API e, opzionalmente, sull'attivita' delle sessioni SSH. Parametri come `IDLE_MINUTES`, `STARTUP_GRACE_MINUTES` e `DRY_RUN` permettono di calibrare o testare la politica senza impatti immediati, rendendo il comportamento prevedibile anche in contesti di VM economiche o risorse limitate.

Infine, lo script `sf_admin_ubuntu.sh` fornisce una console amministrativa interattiva installabile come `sfadmin`. La console integra funzioni di update e manutenzione: gestisce una directory di update, rileva automaticamente artefatti di build (ZIP) e file backend/supporto ammessi, ferma i servizi prima dell'applicazione e crea backup in `/opt/soulframe/backups` per ridurre il rischio in caso di aggiornamenti non riusciti. La stessa console permette di avviare e fermare lo stack tramite `sfctl`, modificare i parametri in `/etc/soulframe/soulframe.env` e `/etc/soulframe/idle.env` e, quando necessario, eseguire lo spegnimento controllato della macchina. Nel complesso, questi servizi di supporto collegano direttamente le esigenze di gestione operativa alla portabilita' dell'architettura: la stessa build WebGL viene riutilizzata in entrambi gli ambienti, mentre cambia soltanto il livello di orchestrazione e il punto di ingresso di rete, come previsto dai requisiti di sistema in Sezione 3.1.

Componente	File Python	Porta	Endpoint chiave	Responsabilita'
Whisper STT	whisper_server.py	8001	GET /health POST /transcribe	Trascrizione Speech-to-Text (STT) di audio caricato (WAV o formati compatibili), gestione lingua e cleanup di file temporanei.
RAG/LLM	rag_server.py	8002	GET /health GET /avatar_stats POST /chat POST /remember POST /recall POST /ingest_file POST /describe_image	Orchestrazione Retrieval-Augmented Generation (RAG): memoria per-avatar su ChromaDB, retrieval ibrido (vettoriale + BM25), chiamate a Ollama per Large Language Model (LLM) ed embedding, ingestione multimodale con OCR per PDF/immagini e descrizione immagini (Gemini se configurato).
Coqui XTTS TTS	coqui_tts_server.py	8004	GET /health POST /tts POST /tts_json POST /set_avatar_voice GET/DELETE /avatar_voice POST /generate_wait_phrases GET /wait_phrase	Sintesi Text-to-Speech (TTS) con voice cloning per avatar, streaming audio e generazione/serving di frasi di attesa; gestione e persistenza dei profili vocali per avatar_id.
Avatar Asset Server	avatar_asset_server.py	8003	GET /health GET /avatars/list POST /avatars/import GET /avatars/id/model.glb DELETE /avatars/id	Import e caching server-side di modelli .glb, deduplicazione via hash URL, self-healing dei metadati e lista unificata di avatar locali di fallback e avatar importati.

Tab. 3.1: Riepilogo dei micro-servizi backend di SOULFRAME: porte, endpoint principali e responsabilita' operative.

4. SVILUPPO DEL PROGETTO: IMPLEMENTAZIONE E CRITICITÀ

4.1 Implementazione frontend

4.1.1 Gestione stati UI e navigazione

Contenuto in preparazione.

4.1.2 Gestione avatar e onboarding con Avaturn

Contenuto in preparazione.

4.1.3 Integrazione Avaturn WebView/SDK nel client Unity

Contenuto in preparazione.

4.1.4 Acquisizione audio e input desktop/touch

Contenuto in preparazione.

4.1.5 Validazione del campione vocale

Contenuto in preparazione.

4.1.6 MainMode conversazionale

Contenuto in preparazione.

4.2 Implementazione backend

4.2.1 Servizio STT

Contenuto in preparazione.

4.2.2 Servizio RAG e memoria per avatar

Contenuto in preparazione.

4.2.3 Servizio TTS e streaming audio

Contenuto in preparazione.

4.2.4 Servizio asset avatar

Contenuto in preparazione.

4.3 Integrazione end-to-end

4.3.1 Orchestrazione richieste tra client, proxy e micro-servizi

Contenuto in preparazione.

4.3.2 Normalizzazione endpoint locale vs produzione

Contenuto in preparazione.

4.3.3 Gestione errori, retry e fallback

Contenuto in preparazione.

4.4 Criticità affrontate e soluzioni

4.4.1 Latenza e timeout

Contenuto in preparazione.

4.4.2 CORS e routing API

Contenuto in preparazione.

4.4.3 OCR e qualità dell'ingestione

Contenuto in preparazione.

4.4.4 Compatibilità dipendenze/modelli

Contenuto in preparazione.

4.4.5 Differenze operative tra Windows e Ubuntu

Contenuto in preparazione.

4.5 Runbook operativo essenziale

Contenuto in preparazione.

4.6 Affidabilità e sicurezza operativa

Contenuto in preparazione.

5. RISULTATI E VALUTAZIONE

5.1 Impostazione della valutazione

5.1.1 Scenari di prova e setup sperimentale

Contenuto in preparazione.

5.1.2 Metriche tecniche adottate

Contenuto in preparazione.

5.1.3 Metriche di esperienza utente

Contenuto in preparazione.

5.2 Risultati tecnici del prototipo

5.2.1 Prestazioni della pipeline STT-RAG-TTS

Contenuto in preparazione.

5.2.2 Latenza end-to-end e stabilità dei servizi

Contenuto in preparazione.

5.2.3 Osservazioni tra ambiente locale e server

Contenuto in preparazione.

5.3 Risultati qualitativi e casi d'uso

5.3.1 Qualità percepita dell'interazione

Contenuto in preparazione.

5.3.2 Usabilità interfaccia desktop e touch

Contenuto in preparazione.

5.3.3 Analisi di casi e failure cases

Contenuto in preparazione.

5.4 Valutazione utenti (estensione facoltativa)

5.4.1 Risultati SUS

Contenuto in preparazione.

5.4.2 Risultati NPS

Contenuto in preparazione.

5.4.3 Confronti tra gruppi

Contenuto in preparazione.

5.5 Discussione dei risultati

5.5.1 Punti di forza

Contenuto in preparazione.

5.5.2 Limiti emersi

Contenuto in preparazione.

5.5.3 Sintesi rispetto alle research questions

Contenuto in preparazione.

6. CONCLUSIONI E SVILUPPI FUTURI

6.1 Sintesi del lavoro svolto

Contenuto in preparazione.

6.2 Contributi principali

Contenuto in preparazione.

6.3 Limiti attuali del sistema

Contenuto in preparazione.

6.4 Sviluppi futuri prioritari

6.4.1 Miglioramenti tecnici del prototipo

Contenuto in preparazione.

6.4.2 Estensione della valutazione utenti

Contenuto in preparazione.

6.5 Considerazioni finali

Contenuto in preparazione.

RINGRAZIAMENTI