

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

| | |
|--|----|
| <i>Abstract</i> | I |
| 1. Introduzione | 1 |
| 1.1 Motivazione e contesto applicativo | 1 |
| 1.2 Perimetro e requisiti di progetto | 3 |
| 1.3 Obiettivi e contributi di SOULFRAME | 4 |
| 1.4 Panoramica del sistema e flusso end-to-end | 6 |
| 1.5 Metodo di lavoro e struttura della tesi | 7 |
| 2. Fondamenti e Stato dell'Arte | 9 |
| 2.1 Agenti conversazionali embodied in XR | 9 |
| 2.1.1 Presenza sociale, co-presenza e ruolo della voce | 9 |
| 2.1.2 Limiti aperti nei sistemi conversazionali immersivi | 10 |
| 2.2 Pipeline AI adottata in SOULFRAME | 11 |
| 2.2.1 Speech-to-Text con Whisper | 12 |
| 2.2.2 Memoria conversazionale con RAG (LLM + embeddings + retrieval) | 13 |
| 2.2.3 Text-to-Speech con Coqui XTTS v2 | 14 |
| 2.3 Posizionamento di SOULFRAME rispetto allo stato dell'arte | 15 |
| 3. Architettura e Tecnologie Utilizzate | 16 |
| 3.1 Requisiti del sistema | 16 |
| 3.1.1 Requisiti funzionali | 16 |
| 3.1.2 Requisiti non funzionali | 17 |
| 3.2 Architettura di riferimento di SOULFRAME | 19 |
| 3.2.1 Vista d'insieme dei componenti frontend/backend | 19 |
| 3.2.2 Flusso end-to-end audio → testo → risposta → audio | 21 |

| | | |
|-------|---|----|
| 3.2.3 | Flusso di gestione avatar (creazione, import, cache, rendering) | 23 |
| 3.3 | Componenti implementati | 27 |
| 3.3.1 | Integrazione Avaturn nel frontend Unity WebGL | 27 |
| 3.3.2 | Backend AI a micro-servizi (Whisper, RAG, TTS, Avatar Asset) | 28 |
| 3.3.3 | Persistenza e gestione dati per avatar | 30 |
| 3.4 | Setup e deploy operativo | 31 |
| 3.4.1 | Ambiente locale Windows | 31 |
| 3.4.2 | Ambiente server Ubuntu | 32 |
| 3.4.3 | Servizi di supporto (systemd, Caddy, script amministrativi) | 33 |
| 4. | <i>Sviluppo del Progetto: Implementazione e Criticità</i> | 35 |
| 4.1 | Implementazione frontend | 35 |
| 4.1.1 | Gestione stati UI e navigazione | 35 |
| 4.1.2 | Gestione avatar e onboarding con Avaturn | 35 |
| 4.1.3 | Integrazione Avaturn WebView/SDK nel client Unity | 35 |
| 4.1.4 | Acquisizione audio e input desktop/touch | 35 |
| 4.1.5 | Validazione del campione vocale | 35 |
| 4.1.6 | MainMode conversazionale | 35 |
| 4.2 | Implementazione backend | 36 |
| 4.2.1 | Servizio STT | 36 |
| 4.2.2 | Servizio RAG e memoria per avatar | 36 |
| 4.2.3 | Servizio TTS e streaming audio | 37 |
| 4.2.4 | Servizio asset avatar | 37 |
| 4.3 | Integrazione end-to-end | 38 |
| 4.3.1 | Orchestrazione richieste tra client, proxy e micro-servizi | 38 |
| 4.3.2 | Normalizzazione endpoint locale vs produzione | 38 |
| 4.3.3 | Gestione errori, retry e fallback | 38 |
| 4.4 | Criticità affrontate e soluzioni | 38 |
| 4.4.1 | Latenza e timeout | 38 |
| 4.4.2 | CORS e routing API | 38 |
| 4.4.3 | OCR e qualità dell'ingestione | 38 |
| 4.4.4 | Compatibilità dipendenze/modelli | 38 |
| 4.4.5 | Differenze operative tra Windows e Ubuntu | 38 |

| | | |
|-------|---|----|
| 4.5 | Runbook operativo essenziale | 38 |
| 4.6 | Affidabilità e sicurezza operativa | 39 |
| 5. | <i>Risultati e Valutazione</i> | 40 |
| 5.1 | Impostazione della valutazione | 40 |
| 5.1.1 | Scenari di prova e setup sperimentale | 40 |
| 5.1.2 | Metriche tecniche adottate | 40 |
| 5.1.3 | Metriche di esperienza utente | 40 |
| 5.2 | Risultati tecnici del prototipo | 40 |
| 5.2.1 | Prestazioni della pipeline STT-RAG-TTS | 40 |
| 5.2.2 | Latenza end-to-end e stabilità dei servizi | 40 |
| 5.2.3 | Osservazioni tra ambiente locale e server | 40 |
| 5.3 | Risultati qualitativi e casi d'uso | 41 |
| 5.3.1 | Qualità percepita dell'interazione | 41 |
| 5.3.2 | Usabilità interfaccia desktop e touch | 41 |
| 5.3.3 | Analisi di casi e failure cases | 41 |
| 5.4 | Valutazione utenti (estensione facoltativa) | 41 |
| 5.4.1 | Risultati SUS | 41 |
| 5.4.2 | Risultati NPS | 41 |
| 5.4.3 | Confronti tra gruppi | 41 |
| 5.5 | Discussione dei risultati | 41 |
| 5.5.1 | Punti di forza | 41 |
| 5.5.2 | Limiti emersi | 41 |
| 5.5.3 | Sintesi rispetto alle research questions | 42 |
| 6. | <i>Conclusioni e Sviluppi Futuri</i> | 43 |
| 6.1 | Sintesi del lavoro svolto | 43 |
| 6.2 | Contributi principali | 43 |
| 6.3 | Limiti attuali del sistema | 43 |
| 6.4 | Sviluppi futuri prioritari | 43 |
| 6.4.1 | Miglioramenti tecnici del prototipo | 43 |
| 6.4.2 | Estensione della valutazione utenti | 43 |
| 6.5 | Considerazioni finali | 43 |

Ringraziamenti 44

Bibliografia 45

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 [1]. 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 [2]. 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 principale [3]. La stessa revisione evidenzia che molte interazioni restano *task-oriented*, ma cresce l'interesse per scambi più dinamici e adattivi grazie a modelli neurali

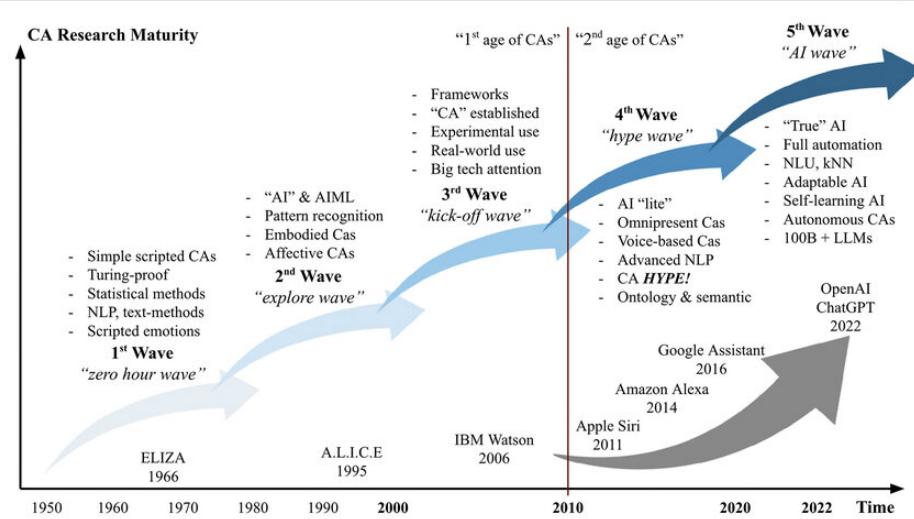


Fig. 1.1: Evoluzione della maturità della ricerca sui Conversational Agents (CA), dalla “zero hour wave” alla “AI wave”, con riferimento ai principali passaggi tecnologici.¹

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 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

¹ Fonte: adattamento da una figura pubblicata su ResearchGate.

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. L’uso di un motore real-time 3D 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 [3]. 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.

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 → RAG/LLM → TTS → output audiovisivo dell’avatar funziona in modo continuo e stabile, includendo anche 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

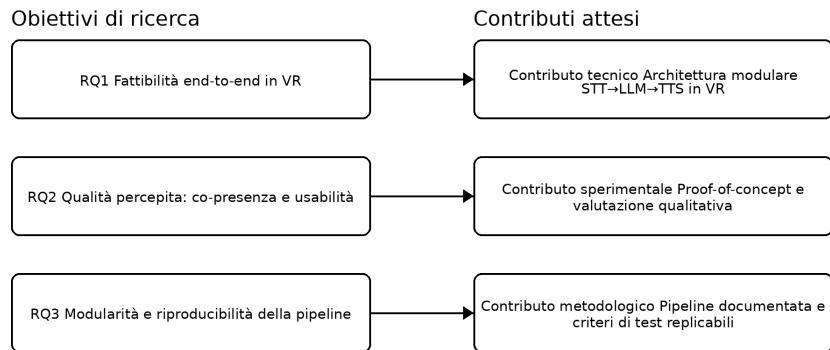


Fig. 1.2: Corrispondenza tra obiettivi di ricerca e contributi del progetto SOULFRAME.

coerenza dell’embodiment, cioè un avatar che allinei voce, aspetto visivo e segnali comportamentali.

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 [2].

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. La sostituzione avviene mantenendo stabili i contratti API tra client e servizi. RQ3 completa i primi due obiettivi, 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. L’architettura 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*. L’impostazione complessiva 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 [4].

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*. Tenere sul client le funzioni sensibili al frame-rate e delegare al backend i moduli più onerosi rende questa separazione operativamente sostenibile.

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 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.

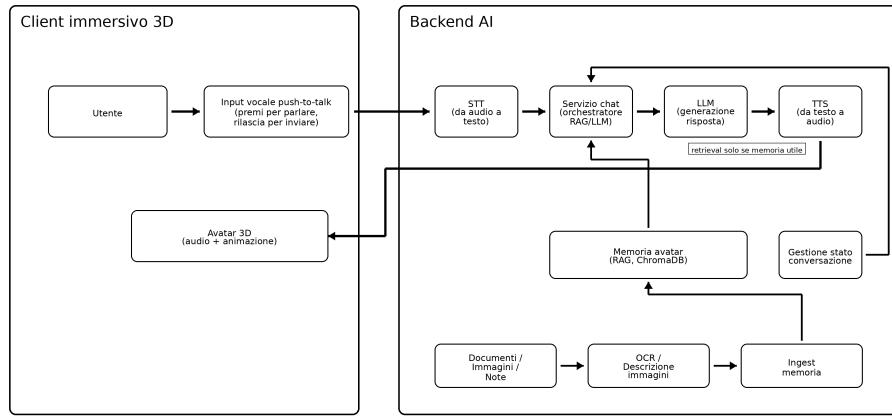


Fig. 1.3: Flusso end-to-end del sistema SOULFRAME: dall’input vocale dell’utente alla risposta dell’agente embodied.

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 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

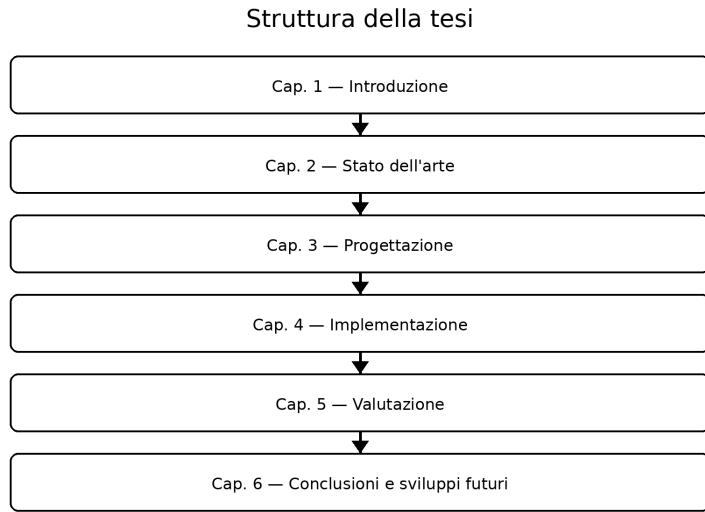


Fig. 1.4: Struttura della tesi e organizzazione dei capitoli.

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.[3] 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 scomponete 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.[5]

La voce è un canale rilevante per la presenza sociale perché rende immediatamente percepibili tempi di risposta, ritmo e prosodia. In VR questo effetto risulta più marcato quando l'agente è embodied: evidenze sperimentali indicano che un

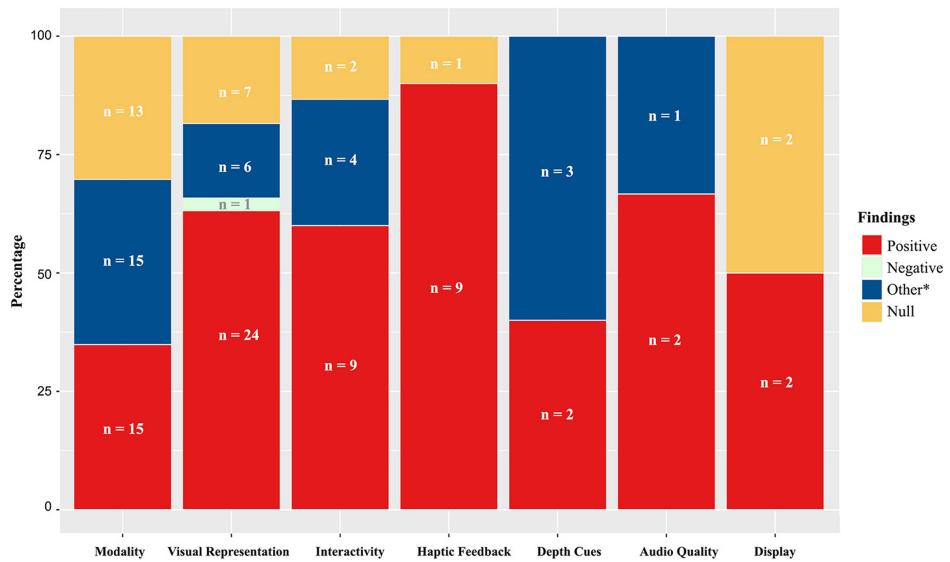


Fig. 2.1: Fattori immersivi associati alla presenza sociale: la qualità audio compare tra le variabili considerate nella letteratura.¹

ECA con interazione vocale in tempo reale (STT+TTS) aumenta la co-presenza percepita rispetto a una condizione con audio pre-registrato.[2] 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 acquisizione audio, trascrizione, generazione e sintesi: ritardi percepibili riducono la naturalezza percepita anche quando il contenuto della risposta è corretto.

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).

Pipeline AI a tre stadi



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

La continuità tra turni è un secondo limite: senza una memoria affidabile il sistema fatica a mantenere riferimenti e preferenze espresse dall’utente. Anche l’integrazione multimodale resta critica, perché richiede coerenza tra risposta linguistica, tempi e segnali non verbali. La valutazione, infine, è complessa: 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.[6] 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. La scomposizione in tre stadi isola i vincoli della conversazione a turni, in particolare la latenza, e permette 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.

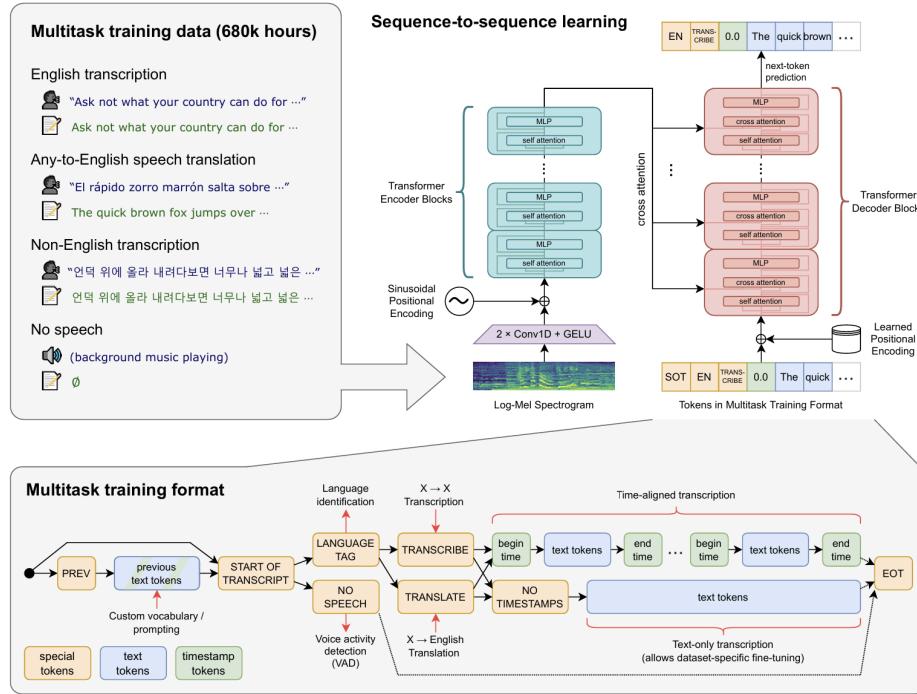


Fig. 2.3: Architettura del modello Whisper: encoder Transformer con log-mel spectrogram in input e decoder per la trascrizione multilingue zero-shot.²

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.[7] Nel prototipo è adottata la versione `medium` come compromesso tra qualità della trascrizione e costo computazionale. La struttura del modello è richiamata in Figura 2.3.

² Fonte: Radford et al., *Robust Speech Recognition via Large-Scale Weak Supervision*, ICML 2023, Fig. 1, <https://arxiv.org/abs/2212.04356>.

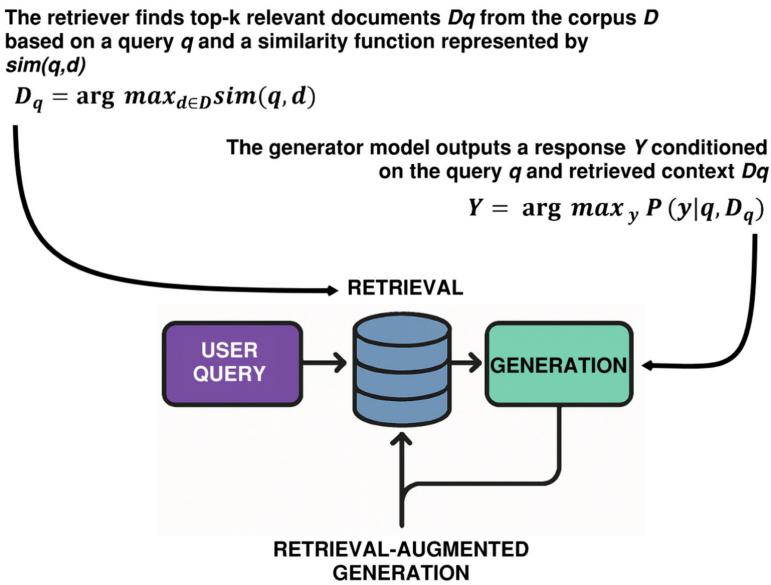


Fig. 2.4: Schema concettuale di Retrieval-Augmented Generation (RAG): recupero di contesto e generazione della risposta.³

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.[8] Figura 2.4 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

³ 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).

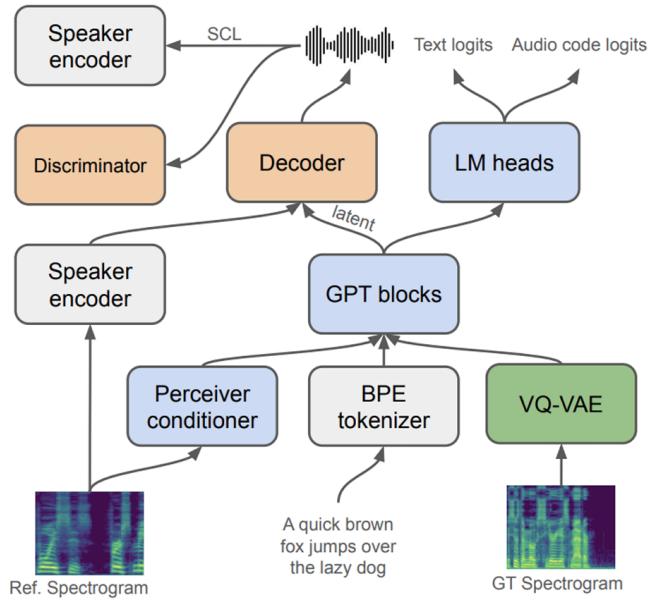


Fig. 2.5: Panoramica dell’architettura XTTS: Conditioning Encoder, encoder GPT-2 con Perceiver Resampler e decoder HiFi-GAN per sintesi multilingue zero-shot con voice cloning.⁴

di documenti (con OCR per PDF) e adotta retrieval ibrido, combinando similarità semantica e segnali lessicali.

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.[9] 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. Un riferimento architettonico del modello è riportato in Figura 2.5.

⁴ Fonte: Casanova et al., *XTTS: a Massively Multilingual Zero-Shot Text-to-Speech Model*, Interspeech 2024, Fig. 1, <https://arxiv.org/abs/2406.04904>.

2.3 Posizionamento di SOULFRAME rispetto allo stato dell'arte

I limiti aperti richiamati nella Sezione 2.1 riguardano soprattutto latenza end-to-end, continuità tra turni, integrazione multimodale e valutazione sperimentale. In SOULFRAME ho scelto una pipeline locale a tre stadi per ridurre dipendenze cloud e mantenere più prevedibile il ciclo audio→testo→risposta→audio. La continuità è sostenuta da una memoria persistente e isolata per avatar, che conserva contesto e preferenze senza interferenze tra profili distinti. Sul piano del retrieval, la combinazione ibrida di segnali semanticici e lessicali rende più robusta la risposta anche con query formulate in modo variabile. Per contenere la latenza percepita durante il turno conversazionale, il sistema usa sintesi in streaming e strategie di attesa che riducono i silenzi udibili. La modularità architetturale consente di sostituire singoli moduli senza modificare il client, favorendo iterazione tecnica e confrontabilità dei risultati.

Il posizionamento non riguarda solo la pipeline: in SOULFRAME l'avatar non è un semplice asset grafico, ma un profilo che combina aspetto, voce e memoria, così da mantenere coerenza tra sessioni e rendere più plausibile la presenza dell'interlocutore. Ho preferito vincolare l'accesso alla conversazione operativa alla disponibilità di un profilo minimo (voce e memoria) per evitare un'esperienza *stateless* in cui ogni turno riparte da zero e l'utente non percepisce progressione. La memoria non è limitata a note manuali: l'ingestione di documenti e immagini consente di trasformare materiali esterni in contesto riutilizzabile, ampliando il perimetro oltre il dialogo puro e rendendo più naturale riferirsi a contenuti che l'utente ha esplicitamente fornito. Sul piano dell'accessibilità, l'esecuzione via browser riduce la barriera di ingresso rispetto a configurazioni VR tradizionali, ma impone vincoli tecnici—acquisizione microfono, gestione dell'audio e latenza—che rendono centrali scelte come streaming, warmup e segnali di attesa. Le implicazioni architettoniche di queste decisioni, insieme ai componenti che le realizzano, vengono discusse nel Capitolo 3, mentre il Capitolo 4 entra nei dettagli implementativi e nei compromessi emersi durante lo sviluppo.

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, disponibilità, modularità).

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 FastAPI¹ 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 è soddisfatto se la lista include almeno un modello locale di fallback (definito in `LOCAL_MODELS`) e, quando il backend è raggiungibile, include anche avatar importati con un URL caricabile del modello `.g1b`.

RF2 (Import e caching asset `.g1b`): quando è disponibile un URL di export (ad es. da Avaturn²), 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 è soddisfatto se import ripetuti dello stesso URL non creano

¹ FastAPI Team, *FastAPI Documentation*, <https://fastapi.tiangolo.com/>, Accessed: 2026-02-22.

² Avaturn, *Avaturn Integration Documentation*, <https://docs.avaturn.me/docs/integration/overview/>, Accessed: 2026-02-22.

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`, associandolo all'avatar attivo; a completamento, il sistema deve poter generare frasi di attesa tramite `/generate_wait_phrases`. Il requisito è soddisfatto se la registrazione supera una verifica di similarità testuale con soglia definita e se il riferimento vocale risulta riutilizzabile nelle sessioni successive; la persistenza è 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 è soddisfatto se la memoria è persistente tra sessioni (finché la directory dati lato server non viene azzerata o cancellata) e se due avatar distinti non condividono documenti indicizzati; l'isolamento può essere verificato confrontando i risultati di `/recall` a parità 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 è 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 multimediale per memoria per-avatar): il sistema deve consentire l'ingestione multimediale nella memoria dell'avatar, includendo descrizione immagine con `/describe_image` e ingestione documentale con `/ingest_file` (estrazione e indicizzazione del contenuto). Il requisito è 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 plausibilità dell'interazione

vocale e la sostenibilità 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 complessità, rendendo utile fissare vincoli di qualità verificabili già in fase architettonica [10].

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 conformità al requisito è 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 [11].

RNF2 (Affidabilità - 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 è 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 conformità al requisito è determinata conteggiando turni riusciti e falliti nei log di sessione.

RNF3 (Manutenibilità - modularity e portabilità): 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 compatibilità (`/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 responsabilità sensibili al frame-rate (interfaccia e resa 3D) e delegare al backend i compiti di inferenza e persistenza. Il client è 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 è scomposta in quattro micro-servizi FastAPI indipendenti, ciascuno con un contratto API mirato. Whisper (porta 8001) espone `/transcribe` e trasforma l’audio in testo. Il servizio RAG (porta 8002) governa l’orchestrazione conversazionale con `/chat`, insieme alla memoria per-avatar (`/remember`, `/recall`) e all’ingestione di contenuti (`/ingest_file`). In questa architettura il RAG è il punto di convergenza: costruisce il contesto via retrieval e delega a un runtime esterno le operazioni di Large Language Model (LLM) ed embedding. Il servizio Text-to-Speech (TTS), basato su Coqui XTTs v2 [9] e in ascolto sulla porta 8004, espone sintesi completa (`/tts`), sintesi in streaming (`/tts_stream`) e endpoint per frasi di attesa (`/wait_phrase`, `/generate_wait_phrases`). L’Avatar Asset Server (porta 8003) gestisce lista e import degli avatar (`/avatars/list`, `/avatars/import`) e serve i modelli `.glb` tramite `/avatars/id/model.glb`, mantenendo persistente la cache degli asset 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 può variare tra ambiente locale e server con GPU dedicata (profilo lightweight vs profilo standard), mentre il modello embedding resta dedicato al retrieval. L’uso di Ollama come runtime separato 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

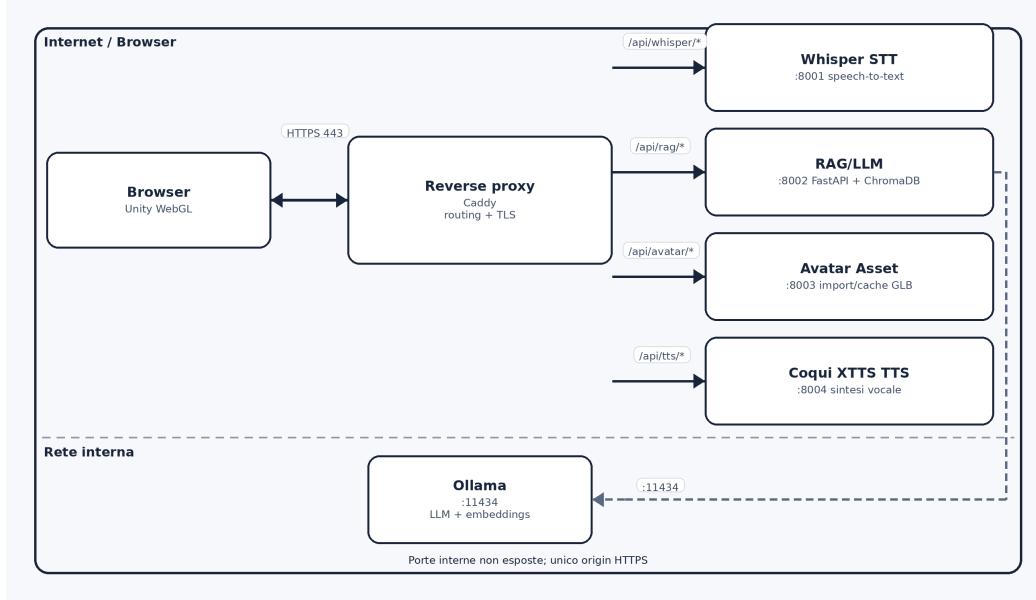


Fig. 3.1: Vista d'insieme dell'architettura a componenti di SOULFRAME: il client Unity WebGL comunica con i micro-servizi backend attraverso il reverse proxy Caddy.

endpoint e formati attesi. I dettagli di confronto tra configurazioni modello sono discussi nel Capitolo 4.

In deploy pubblico, l'elemento chiave di integrazione è il reverse proxy Caddy, che svolge due funzioni: terminazione TLS e punto d'ingresso unico per il browser. Il client WebGL comunica con un solo origin HTTPS e invoca le API tramite path riscritti (`/api/whisper/*`, `/api/rag/*`, `/api/avatar/*`, `/api/tts/*`); Caddy inoltra le richieste alle porte interne 8001–8004, mantenute non esposte verso l'esterno. Il reverse proxy riduce la complessità 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 responsabilità 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 disponibilità delle

porte) per ridurre conflitti tra processi durante lo sviluppo. In ambiente Ubuntu, ciascun micro-servizio è gestito come unità `systemd`, con comandi amministrativi che permettono start/stop/status e aggiornamenti senza dover riconfigurare manualmente l'intero stack. Nel lavoro di sviluppo ho mantenuto questa separazione perché i colli di bottiglia più difficili da diagnosticare emergevano tra servizi diversi, non dentro un singolo modulo.

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

Il flusso conversazionale end-to-end di SOULFRAME è 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 responsabilità di interazione e rendering e delegando al backend le fasi di inferenza. L'interazione è 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, così da appoggiarsi alle primitive audio disponibili nel runtime WebGL.³

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"`. L'isolamento dell'Automatic Speech Recognition (ASR) in un componente dedicato rende esplicita la prima trasformazione del flusso, da segnale audio (WAV) a contenuto testuale (stringa), su cui è poi possibile applicare logiche conversazionali e di memoria.

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 (ChromaDB⁴) e popolata tramite embedding generati con un modello dedicato (nella confi-

³ Unity Technologies, *Audio in WebGL*, <https://docs.unity3d.com/6000.2/Documentation/Manual/webgl-audio.html>, Accessed: 2026-02-22.

⁴ Chroma, *Chroma Documentation*, <https://docs.trychroma.com/>, Accessed: 2026-02-22.

gurazione 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 variabilità 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, perché sposta l'attesa dal completamento dell'intero file audio alla sola generazione dell'incipit, migliorando la continuità percepita anche quando la risposta è lunga.

SOULFRAME introduce inoltre strategie specifiche per mitigare i tempi morti più evidenti. Per contenere la latenza iniziale, il servizio TTS esegue un warmup durante il boot: 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 modalità operative solo quando il TTS risulta pronto. Lo stesso problema può riemergere durante i turni successivi, quando la pipeline STT→RAG/LLM→TTS è in corso: in questa fase il sistema può riprodurre brevi vocalizzi o frasi di attesa pre-generate (ad esempio “hm” o “un secondo”) per evitare silenzi prolungati e dare un segnale immediato di reattività. Le wait phrases sono associate al profilo vocale dell'avatar e vengono generate una tantum o rigenerate quando necessario; dal punto di vista architettonico costituiscono un canale asincrono di feedback che non altera il contenuto della risposta, ma agisce sulla percezione temporale dell'utente.

Figura 3.3 sintetizza la sequenza temporale e i formati dati scambiati tra

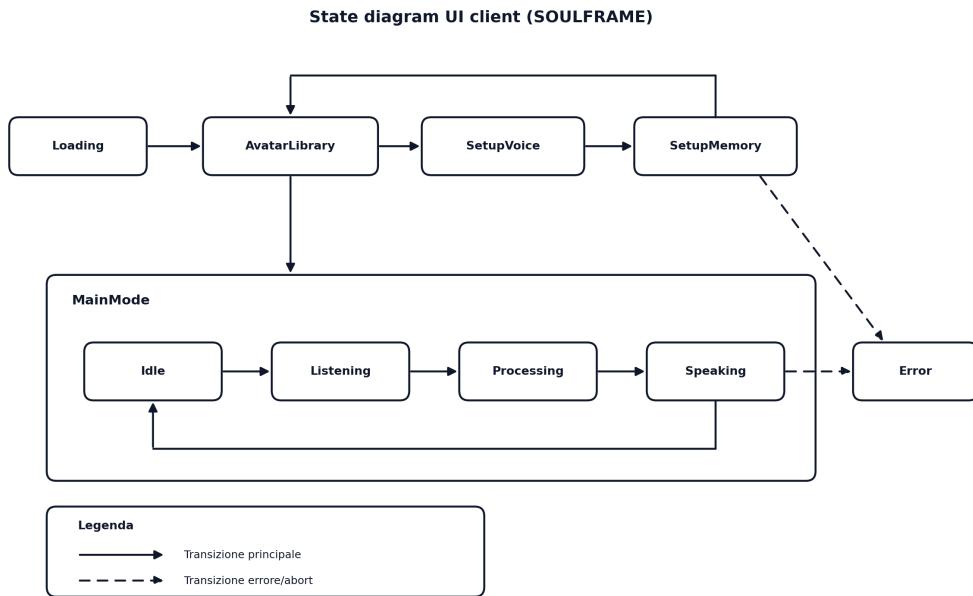


Fig. 3.2: Diagramma degli stati UI del client SOULFRAME: dalla fase di caricamento iniziale al ciclo conversazionale (Idle → Listening → Processing → Speaking) e alla gestione degli errori.

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 è 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 architettonale è 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 servire asset esterni in formato .glb senza dipendere direttamente da URL remoti durante il rendering.

Gli avatar locali costituiscono il percorso più semplice: i modelli LOCAL_model1

Flusso end-to-end audio → testo → risposta → audio (SOULFRAME)

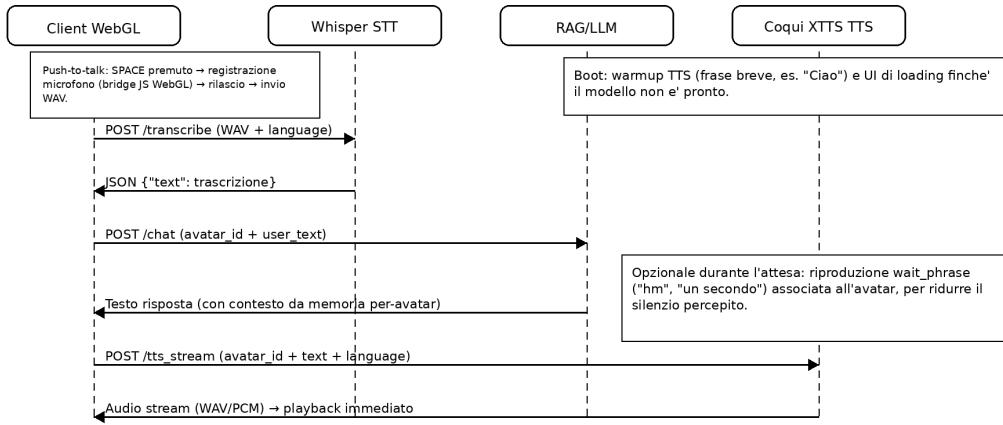


Fig. 3.3: Flusso end-to-end di una richiesta conversazionale in SOULFRAME: dall’acquisizione audio push-to-talk alla riproduzione della risposta vocale.

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 modalità non è necessario alcun passaggio di import o caching, poiché l’asset è già 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 interoperabilità Unity WebGL tra JavaScript e C# basato su

plugin `.jslib` e chiamate di messaggistica verso `GameObject`.⁵ 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. La cache server-side che ne risulta libera il client dalla dipendenza dall'URL originario di Avaturn durante le sessioni successive: il modello viene caricato da un endpoint controllato e coerente con il deploy. Il modello viene poi esposto tramite `GET /avatars/{id}/model.glb` e pubblicato al client come `cached_glb_url` all'interno della risposta di `/avatars/list`, che aggredisce sempre avatar locali e importati in un'unica libreria.

Un aspetto rilevante di robustezza è 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 probabilità 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, così che la transizione tra profili sia percepita come un'operazione di selezione indipendente dalla provenienza (locale o importata). Figura 3.4 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

⁵ Unity Technologies, *Interaction with browser scripting in WebGL*, <https://docs.unity3d.com/6000.2/Documentation/Manual/webgl-interacting-browser-js.html>, Accessed: 2026-02-22.

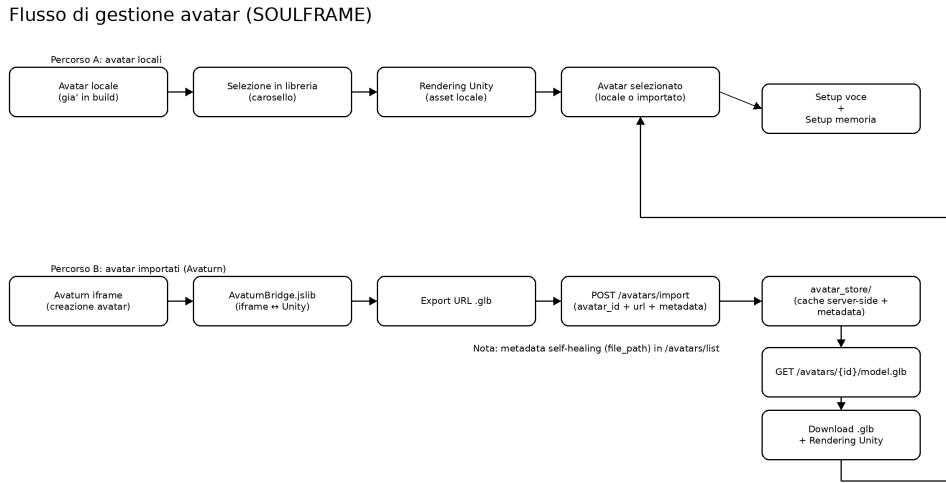


Fig. 3.4: Flusso di gestione avatar in SOULFRAME: dalla creazione tramite Avaturn alla cache server-side e al rendering nel client Unity.

frase attesa mostrata in UI, applicando una metrica di similarità 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 è assente, il client indirizza a `SetupMemory`, che offre tre modalità: 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 già dai primi turni di `/chat`; la presenza dei contenuti è verificabile sia con `/recall` sia con `/avatar_stats`. I dettagli implementativi della pipeline di estrazione e indicizzazione sono discussi nel Capitolo 4. In termini di stato applicativo, il client può 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. Alcune schermate operative del client sono riportate in Figura 3.5, Figura 3.6 e Figura 3.7.



Fig. 3.5: Interfaccia carosello avatar in SOULFRAME: selezione del profilo attivo tra avatar locali e importati.

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 è chiarire quali moduli concretizzano i flussi discussi nel capitolo, senza entrare nei dettagli più minimi 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; la panoramica unificata dei micro-servizi backend è riportata in Tabella 3.1.

3.3.1 Integrazione Avaturn nel frontend Unity WebGL

L’integrazione di Avaturn nel frontend è realizzata incapsulando l’editor di personalizzazione in un overlay iframe aperto dal client Unity WebGL. La gestione del ciclo di vita dell’overlay (apertura, ricezione evento export, chiusura) è delegata a un bridge JavaScript che inoltra al runtime Unity un payload JSON con URL del modello .glb e metadati dell’avatar tramite il meccanismo stan-



Fig. 3.6: Stato di ascolto push-to-talk nel client Unity WebGL: acquisizione audio attiva con indicatore visivo.

dard di messaggistica .jslib.⁶ Il client riceve il JSON e delega la fase di import al backend; da quel momento il flusso converge sul percorso descritto in Sezione 3.2. I dettagli implementativi del bridge (gestione del DOM, interoperabilità C#/JavaScript, compatibilità con il prefab Avaturn) sono discussi nel Capitolo 4.

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

Il backend di SOULFRAME è realizzato come insieme di micro-servizi indipendenti basati su FastAPI, avviati con `uvicorn` su porte dedicate (8001–8004). La scomposizione in servizi rende esplicita la separazione delle responsabilità e consente di verificare disponibilità e corretto instradamento con controlli semplici (`/health`) prima di abilitare il flusso conversazionale. Tabella 3.1 sintetizza i quattro componenti e i relativi contratti API: il riepilogo è utile sia in fase di sviluppo (per individuare rapidamente dipendenze e punti di failure) sia per la lettura del capitolo, perché connette porte, endpoint e responsabilità operative in un'unica vista.

⁶ Unity Technologies, *Interaction with browser scripting in WebGL*, <https://docs.unity3d.com/6000.2/Documentation/Manual/webgl-interacting-browser-js.html>, Accessed: 2026-02-22.



Fig. 3.7: MainMode conversazionale in SOULFRAME: trascrizione e risposta dell'avatar in corso durante un turno vocale.

Whisper presidia la trasformazione audio→testo e costituisce il punto di ingresso del canale vocale utente. Il servizio RAG/LLM governa memoria per-avatar e generazione contestuale, coordinando retrieval e risposta conversazionale. Il servizio TTS converte la risposta in audio, riusando profili vocali per-avatar e gestendo le frasi di attesa. L'Avatar Asset Server isola il ciclo di vita dei modelli .glb (import, cache e distribuzione) dal resto della pipeline conversazionale. In questa sottosezione, il termine *ricerca ibrida* indica la combinazione di similarità vettoriale e BM25 nella fase di retrieval del RAG.

La scomposizione in servizi autonomi mantiene il client Unity disaccoppiato dalla logica interna di inferenza e storage, rendendo più semplice evolvere singoli componenti senza cambiare il flusso utente. In pratica ho mantenuto questo vincolo per poter aggiornare STT, RAG o TTS in modo indipendente durante i test comparativi. I dettagli implementativi (caricamento modelli, gestione file temporanei, chunking/overlap, estrazione contenuto, deduplicazione, self-healing e parametri runtime) sono discussi nel Capitolo 4. La documentazione automatica degli endpoint è riportata in Figura 3.8.

| Componente | File Python | Porta | Endpoint chiave | Responsabilità |
|---------------------|------------------------|-------|--|---|
| 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 (similarità vettoriale + BM25), chiamate a Ollama per LLM/embedding e ingestione multimodale con estrazione/indicizzazione dei contenuti. |
| 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 responsabilità operative.

3.3.3 Persistenza e gestione dati per avatar

La persistenza dei dati in SOULFRAME è organizzata su filesystem e segue una separazione per `avatar_id`: ogni profilo mantiene in modo indipendente tre store dedicati, ovvero `avatar_store/` per gli asset .gltf, `voices/avatars/` per il profilo vocale e `rag_store/` per la memoria conversazionale. Questa struttura mantiene allineati identità dell'avatar e risorse operative lungo tutto il ciclo di vita del profilo.

L'isolamento per profilo consente di evitare contaminazioni tra avatar distinti e di ricostruire lo stato dopo riavvio rileggendo i rispettivi percorsi persistenti. A livello applicativo, la presenza di memoria può essere verificata tramite `/avatar_stats` (campo `has_memory`), mentre asset e voce restano associati allo stesso `avatar_id` nei rispettivi store.

I dettagli implementativi (deduplicazione, scrittura atomica, self-healing dei



Fig. 3.8: Interfaccia Swagger UI (/docs) del servizio RAG di SOULFRAME: gli endpoint esposti da `rag_server.py` con schema automatico generato da FastAPI.

metadati, generazione lazy delle frasi di attesa) sono discussi nel Capitolo 4.

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 portabilità e manutenibilità discussi in Sezione 3.1. Per ridurre errori di esercizio, ho privilegiato passaggi operativi ripetibili e un punto di controllo unificato per i servizi.

3.4.1 Ambiente locale Windows

La configurazione locale è orientata a sviluppo e test rapidi: i servizi girano su loopback e la build Unity WebGL è servita in HTTP. Il provisioning iniziale è automatizzato da `setup_soulframe_windows.bat`, che prepara ambiente Python, dipendenze backend e configurazioni base necessarie all’avvio del prototipo. In questa modalità l’obiettivo non è la hardening dell’infrastruttura, ma la riduzione del tempo tra modifica del codice e verifica end-to-end.

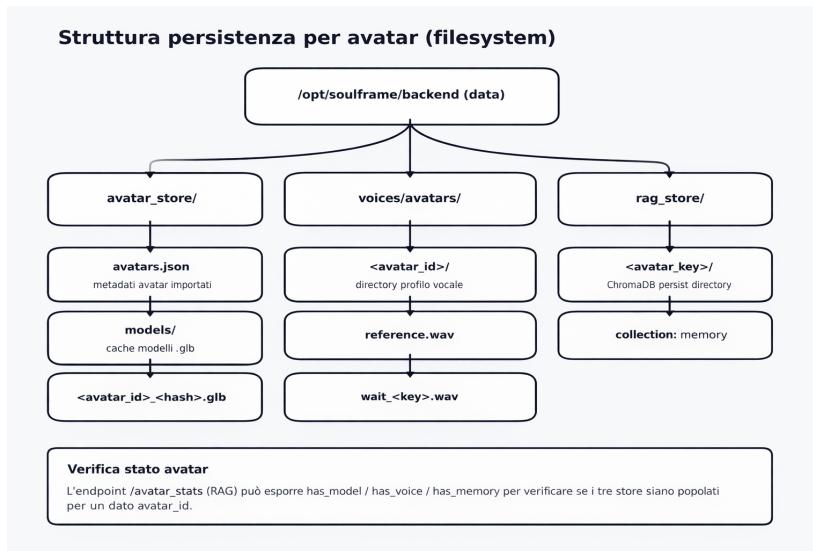


Fig. 3.9: Struttura del filesystem di persistenza in SOULFRAME: tre store isolati per `avatar_id` (asset `.glb`, profilo vocale e memoria RAG).

L'avvio operativo è centralizzato in `ai_services.cmd`, che coordina i processi applicativi, riduce i conflitti su porte già occupate e applica una sequenza di start coerente con le dipendenze della pipeline. In pratica, lo sviluppatore mantiene un punto unico per start/stop/restart dello stack locale, evitando avvii manuali separati dei micro-servizi.

Sul piano funzionale, questo assetto favorisce debugging iterativo e osservabilità immediata: i log dei servizi sono disponibili in console, gli endpoint risultano interrogabili direttamente su loopback e il client WebGL può essere validato senza introdurre variabili infrastrutturali esterne. Le differenze operative rispetto al deploy server sono sintetizzate in Tabella 3.2.

3.4.2 Ambiente server Ubuntu

In ambiente Ubuntu il deploy è orientato all'esposizione pubblica controllata del prototipo. I micro-servizi restano su rete interna, mentre il browser raggiunge un unico origin HTTPS tramite reverse proxy. L'adozione di un origin HTTPS unico semplifica l'integrazione lato client (stessa origine, path applicativi uniformi) e riduce la superficie di esposizione diretta delle singole porte interne.

Il setup server organizza artefatti e configurazione runtime in percorsi dedi-

| Aspetto | Windows locale | Ubuntu server |
|-----------------------------|--|--|
| Avvio servizi | Script locale (<code>ai_services.cmd</code>) con processi separati in ascolto su loopback | Service manager con unità dedicate e restart automatico |
| Origin/URL usati dal client | Origin locale (<code>http://localhost:8000</code>) con porte backend esplicite in configurazione client | Origin HTTPS unico; API raggiunte via path <code>/api/<servizio></code> dietro reverse proxy |
| TLS | Assente nel loop di sviluppo locale (HTTP) | Terminazione TLS al reverse proxy |
| Logging | Log di console/processo per debugging interattivo | Log centralizzati di servizi e reverse proxy |
| Update/rollback | Aggiornamento manuale dello workspace e riavvio script | Script amministrativi con stop/start orchestrato e supporto backup/rollback |
| Gestione persistenze | Directory locali del progetto (<code>avatar_store</code> , <code>voices</code> , <code>rag_store</code>) | Directory persistenti lato server con separazione per componente/avatar |

Tab. 3.2: Confronto operativo tra setup locale Windows e deploy server Ubuntu.

cati, separando componenti applicativi, file statici WebGL e variabili ambientali operative. La gestione dei processi è demandata a unità del service manager con politiche di restart, così che il ripristino dopo reboot o failure non richieda intervento manuale su ogni micro-servizio.

Dal punto di vista del routing, il reverse proxy gestisce terminazione TLS, serving dei file statici e inoltro delle richieste API verso i servizi interni. In questo modo la stessa build Unity WebGL resta riutilizzabile tra locale e server, variando principalmente il livello di orchestrazione e il punto di ingresso di rete. Anche in questo caso i dettagli puntuali di provisioning e parametrizzazione sono rinviati al Capitolo 4.

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

Accanto ai micro-servizi applicativi, il prototipo include un livello di supporto operativo che copre tre esigenze: lifecycle dei processi, gateway/reverse proxy e automazione amministrativa. Questo livello è cruciale per mantenere ripetibile la gestione dell'ambiente, soprattutto quando la piattaforma viene aggiornata o riavviata frequentemente.

Il service manager fornisce avvio al boot, restart automatico e consultazione unificata dei log, consentendo interventi mirati su singoli servizi o sull'intero stack. Il reverse proxy concentra invece le responsabilità di ingress (TLS, routing API e serving statico), così da mantenere il browser disaccoppiato dai dettagli di rete interni.

Gli script amministrativi completano il quadro con procedure standardizzate di manutenzione (stato, restart, aggiornamento, backup e rollback), riducendo il rischio di azioni manuali non ripetibili. La tabella comparativa precedente rende verificabili queste differenze tra ambienti senza appesantire la lettura del capitolo; i dettagli esecutivi sono approfonditi nel Capitolo 4.

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

Il servizio STT è implementato in `whisper_server.py` e viene esposto tramite FastAPI su porta dedicata. All'avvio, il modello viene caricato una sola volta con `whisper.load_model`, guidato dalla variabile d'ambiente `WHISPER_MODEL` (default: `small`), così da evitare costi di inizializzazione a ogni richiesta.

L'endpoint principale è `POST /transcribe`: riceve un `UploadFile` audio e il parametro `language`, salva temporaneamente il payload su disco (con file temporaneo), invoca `model.transcribe` e restituisce una risposta minimale nel campo `text`. La gestione dei file temporanei è protetta da cleanup esplicito in blocco `finally`, per evitare accumulo di artefatti in caso di errori o richieste interrotte.

4.2.2 Servizio RAG e memoria per avatar

Il servizio RAG è implementato in `rag_server.py` e centralizza orchestrazione conversazionale, memoria per-avatar e ingestione contenuti. La configurazione runtime è demandata a variabili d'ambiente (`OLLAMA_HOST`, `CHAT_MODEL`, `EMBED_MODEL`, `RAG_DIR`), così da separare codice e profilo di deploy.

Nel percorso `POST /chat`, il server usa `avatar_id` per risolvere lo store dell'avatar, calcola embedding query via Ollama (`/api/embed`), esegue retrieval, compone il prompt e invoca la generazione (`/api/chat`). La persistenza è isolata per avatar in sottodirectory dedicate sotto `rag_store/`; la collezione viene istanziata con `get_or_create_collection(name="memory")`.

La memoria applicativa include `/remember` per note testuali, `/recall` per diagnostica e `/avatar_stats` per segnali operativi (`count`, `has_memory`). Il retrieval è ibrido: similarità vettoriale su ChromaDB + ranking lessicale BM25, con combinazione pesata tramite `bm25_weight`.

L'ingestione file (`/ingest_file`) applica una pipeline per estensione: PDF con estrazione testo/OCR (PyMuPDF + OCR), immagini con OCR (`pytesseract`), file testuali come plain text. Il contenuto viene segmentato con parametri `RAG_CHUNK_CHARS` e `RAG_CHUNK_OVERLAP`, deduplicato a livello chunk e indicizzato tramite embedding in batch. L'endpoint `/describe_image` abilita descrizione multimodale (Gemini, se configurato) con eventuale salvataggio in memoria.

4.2.3 Servizio TTS e streaming audio

Il servizio TTS è implementato in `coqui_tts_server.py` e usa Coqui XTTs v2 con profilo configurabile da ambiente (`COQUI_TTS_MODEL`, `COQUI_LANG`, `COQUI_AVATAR_VOICES_DIR`). La voce per-avatar viene registrata con `POST/set_avatar_voice`, che salva il riferimento in `voices/avatars/<avatar_id>/reference.wav`; `GET/DELETE/avatar_voice` supportano verifica e reset.

Per la sintesi, il servizio espone sia output streaming (`/tts`, `/tts_stream`) sia formato JSON/base64 (`/tts_json`) per client che non gestiscono stream audio. Le frasi di attesa sono gestite con `/generate_wait_phrases` e `/wait_phrase`, con persistenza per-avatar e possibilità di rigenerazione lazy se manca una clip.

Il bootstrap include parametri di warmup (`COQUI_WARMUP_ON_STARTUP`, `COQUI_WARMUP_TEXT`) per ridurre il costo della prima richiesta reale.

4.2.4 Servizio asset avatar

Il servizio asset è implementato in `avatar_asset_server.py` e fornisce lista, import e serving dei modelli .gltf. `/avatars/list` unifica modelli locali di fallback (`LOCAL_MODELS`) e avatar importati, esponendo per questi ultimi un `cached_gltf_url` coerente con il contesto di deploy.

L'import (`/avatars/import`) valida l'URL sorgente, calcola hash SHA-256 (`url_hash`) e applica deduplicazione: se un record equivalente è già presente e risolvibile su disco, il servizio evita il redownload. In caso di nuovo import, il file viene salvato in `avatar_store/models/` e i metadati sono aggiornati in `avatar_store/avatars.json`.

La persistenza è resa robusta da scritture atomiche su metadati (file temporaneo + replace), download con file `.part`, e logica di self-healing del path (`resolve_avatar_file_path`) per ricostruire collegamenti validi dopo migrazioni o metadati obsoleti. L'asset finale è servito tramite `GET/avatars/{avatar_id}/model.gltf`; `DELETE/avatars/{avatar_id}` gestisce la rimozione controllata.

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

BIBLIOGRAFIA

- [1] Mel Slater e Maria V. Sanchez-Vives. «Enhancing Our Lives with Immersive Virtual Reality». In: *Frontiers in Robotics and AI* 3 (2016), p. 74. DOI: [10.3389/frobt.2016.00074](https://doi.org/10.3389/frobt.2016.00074). URL: <https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2016.00074/full>.
- [2] Peter Kán, Martin Rumpelnik e Hannes Kaufmann. «Embodied Conversational Agents with Situation Awareness for Training in Virtual Reality». In: *ICAT-EGVE 2023 – International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments*. The Eurographics Association, 2023, pp. 27–36. DOI: [10.2312/egve.20231310](https://doi.org/10.2312/egve.20231310). URL: <https://diglib.eg.org/bitstream/handle/10.2312/egve20231310/027-036.pdf>.
- [3] Fu-Chia Yang et al. «Embodied Conversational Agents in Extended Reality: A Systematic Review». In: *IEEE Access* 13 (2025), pp. 79805–79824. DOI: [10.1109/ACCESS.2025.3566698](https://doi.org/10.1109/ACCESS.2025.3566698). URL: <https://doi.org/10.1109/ACCESS.2025.3566698>.
- [4] Liliana Laranjo et al. «Conversational agents in healthcare: a systematic review». In: *Journal of the American Medical Informatics Association* 25.9 (2018). Free full text via PubMed Central (PMCID: PMC6118869), pp. 1248–1258. DOI: [10.1093/jamia/ocy072](https://doi.org/10.1093/jamia/ocy072). URL: <https://pmc.ncbi.nlm.nih.gov/articles/PMC6118869/>.
- [5] Catherine S. Oh, Jeremy N. Bailenson e Gregory F. Welch. «A Systematic Review of Social Presence: Definition, Antecedents, and Implications». In: *Frontiers in Robotics and AI* 5 (2018), p. 114. DOI: [10.3389/frobt.2018.00114](https://doi.org/10.3389/frobt.2018.00114). URL: <https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2018.00114/full>.

- [6] Michele Yin et al. «Let’s Give a Voice to Conversational Agents in Virtual Reality». In: *Interspeech 2023*. 2023, pp. 5247–5248. URL: https://www.isca-archive.org/interspeech_2023/yin23b_interspeech.pdf.
- [7] Alec Radford et al. «Robust Speech Recognition via Large-Scale Weak Supervision». In: *Proceedings of the 40th International Conference on Machine Learning (ICML)*. Vol. 202. Proceedings of Machine Learning Research. Also available on arXiv: <https://arxiv.org/abs/2212.04356>. PMLR, 2023, pp. 28492–28518. URL: <https://proceedings.mlr.press/v202/radford23a.html>.
- [8] Patrick Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». In: *Advances in Neural Information Processing Systems*. Vol. 33. Also available on arXiv: <https://arxiv.org/abs/2005.11401>. 2020, pp. 9459–9474. URL: <https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>.
- [9] Edresson Casanova et al. «XTTS: a Massively Multilingual Zero-Shot Text-to-Speech Model». In: *Interspeech 2024*. 2024, pp. 4978–4982. DOI: [10.21437/Interspeech.2024-2016](https://doi.org/10.21437/Interspeech.2024-2016). URL: https://www.isca-archive.org/interspeech_2024/casanova24_interspeech.pdf.
- [10] Sai Anirudh Karre e Y. Raghu Reddy. «Model-based approach for specifying requirements of virtual reality software products». In: *Frontiers in Virtual Reality* 5 (2024). Open Access. DOI: [10.3389/frvir.2024.1471579](https://doi.org/10.3389/frvir.2024.1471579). URL: <https://www.frontiersin.org/journals/virtual-reality/articles/10.3389/frvir.2024.1471579/full>.
- [11] Spencer Lin et al. «Estuary: A Framework For Building Multimodal Low-Latency Real-Time Socially Interactive Agents». In: *Proceedings of the ACM International Conference on Intelligent Virtual Agents (IVA 2024)*. Also available on arXiv: <https://arxiv.org/abs/2410.20116>. ACM, 2024. DOI: [10.1145/3652988.3696198](https://doi.org/10.1145/3652988.3696198). URL: <https://dl.acm.org/doi/10.1145/3652988.3696198>.