

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 conversazionale per
mitigare la loneliness

Integrazione di agenti virtuali per il sostegno alla socialità

RELATORE

Prof.ssa Paola Barra

CANDIDATO

Luca Tartaglia

Matr. 0124002294

Anno Accademico 2024/2025

ABSTRACT

SOULFRAME nasce da un'esigenza molto concreta: in tanti ambienti VR la scena è curata e credibile, ma quando si prova a parlare con l'agente l'esperienza si rompe subito, tra risposte poco coerenti, tempi d'attesa e interazioni macchinose. Questa tesi presenta SOULFRAME, un prototipo di sistema conversazionale con avatar 3D pensato per affrontare il tema della loneliness e, soprattutto, per rendere il dialogo con l'AI più naturale e accessibile: l'utente deve poter parlare senza "smanettare", con un'interfaccia chiara e adatta sia a desktop (tastiera e controlli tradizionali) sia a mobile/touch (push-to-talk e navigazione semplificata).

Il progetto è costruito con una logica client-server: Unity WebGL gestisce avatar, interfaccia e registrazione dell'audio; sul backend una serie di servizi separati si occupa di capire la voce, generare risposte contestuali con supporto di memoria e restituire una voce sintetica credibile. La memoria è legata al singolo avatar e può essere arricchita con note e contenuti esterni (anche documenti e immagini).

Durante lo sviluppo sono emersi punti critici tipici di un sistema end-to-end: integrazione tra componenti, latenza percepita e compatibilità tra ambienti. Per ridurre attriti e errori, ho dato molta importanza anche all'automazione di setup e gestione dei servizi. Il risultato è un prototipo stabile, utilizzabile sia in locale sia su server, che rende chiari i colli di bottiglia e mostra una pipeline vocale davvero impiegabile in scenari realistici.

INDICE

<i>Abstract</i>	I
<i>Elenco delle figure</i>	VI
<i>Elenco delle tabelle</i>	X
1. Introduzione	1
1.1 Motivazione e contesto applicativo	1
1.2 Perimetro e requisiti di progetto	2
1.3 Obiettivi e contributi di SOULFRAME	2
1.4 Panoramica del sistema e flusso end-to-end	3
1.5 Metodo di lavoro e struttura della tesi	4
2. Fondamenti e Stato dell'Arte	6
2.1 Agenti conversazionali embodied in XR	6
2.1.1 Presenza sociale, co-presenza e ruolo della voce	6
2.1.2 Limiti aperti nei sistemi conversazionali immersivi	8
2.2 Pipeline AI adottata in SOULFRAME	9
2.2.1 Speech-to-Text con Whisper	9
2.2.2 Memoria conversazionale con RAG (LLM + embeddings + retrieval)	10
2.2.3 Text-to-Speech con Coqui XTTS v2	11
2.3 Posizionamento di SOULFRAME rispetto allo stato dell'arte	12
3. Architettura e Tecnologie Utilizzate	14
3.1 Requisiti del sistema	14
3.1.1 Requisiti funzionali	14
3.1.2 Requisiti non funzionali	15

3.2	Architettura di riferimento di SOULFRAME	17
3.2.1	Vista d'insieme dei componenti frontend/backend	17
3.2.2	Flusso end-to-end audio → testo → risposta → audio . . .	19
3.2.3	Flusso di gestione avatar (creazione, import, cache, rendering)	22
3.3	Componenti implementati	23
3.3.1	Integrazione Avaturn nel frontend Unity WebGL	23
3.3.2	Backend AI a micro-servizi (Whisper, RAG, TTS, Avatar Asset)	24
3.3.3	Persistenza e gestione dati per avatar	26
3.4	Setup e deploy operativo	26
3.4.1	Ambiente locale Windows	27
3.4.2	Ambiente server Ubuntu	28
3.4.3	Servizi di supporto (systemd, Caddy, script amministrativi)	29
4.	<i>Sviluppo del Progetto: Implementazione e Criticità</i>	33
4.1	Implementazione frontend	33
4.1.1	Gestione stati UI e navigazione	33
4.1.2	Gestione avatar e onboarding con Avaturn	36
4.1.3	Integrazione Avaturn WebView/SDK nel client Unity . . .	38
4.1.4	Acquisizione audio e input desktop/touch	39
4.1.5	Validazione del campione vocale	40
4.1.6	Comportamento embodied: lip sync e sguardo idle	45
4.1.7	Post-processing, rings di stato e feedback di selezione . .	46
4.1.8	MainMode conversazionale	47
4.2	Implementazione backend	49
4.2.1	Servizio STT (Whisper)	50
4.2.2	Servizio RAG e memoria per avatar	51
4.2.3	Servizio TTS e streaming audio	55
4.2.4	Servizio asset avatar	58
4.3	Integrazione end-to-end	60
4.3.1	Orchestrazione richieste tra client, proxy e micro-servizi .	60
4.3.2	Normalizzazione endpoint locale vs produzione	61
4.3.3	Gestione errori, retry e fallback	62
4.4	Criticità affrontate e soluzioni	63

4.4.1	Latenza e timeout	63
4.4.2	CORS e routing API	65
4.4.3	OCR e qualità dell'ingestione	66
4.4.4	Compatibilità dipendenze/modelli	67
4.4.5	Differenze operative tra Windows e Ubuntu	68
4.5	Runbook operativo essenziale	69
4.6	Affidabilità e sicurezza operativa	75
5.	<i>Risultati e Valutazione</i>	81
5.1	Impostazione della valutazione	81
5.1.1	Scenari di prova e setup sperimentale	81
5.1.2	Metriche tecniche adottate	82
5.1.3	Metriche di esperienza utente	83
5.2	Risultati tecnici del prototipo	83
5.2.1	Prestazioni della pipeline STT-RAG-TTS	83
5.2.2	Latenza end-to-end e stabilità dei servizi	84
5.2.3	Osservazioni tra ambiente locale e server	85
5.3	Risultati qualitativi e casi d'uso	86
5.3.1	Qualità percepita dell'interazione	86
5.3.2	Usabilità interfaccia desktop e touch	88
5.3.3	Analisi di casi e failure cases	90
5.4	Discussione dei risultati	93
5.4.1	Punti di forza	93
5.4.2	Criticità osservate	94
5.4.3	Limiti emersi	95
5.4.4	Sintesi rispetto alle research questions	96
6.	<i>Conclusioni e Sviluppi Futuri</i>	98
6.1	Sintesi del lavoro svolto	98
6.2	Contributi principali	98
6.3	Limiti attuali del sistema	99
6.4	Sviluppi futuri prioritari	101
6.4.1	Miglioramenti tecnici del prototipo	101
6.4.2	Estensione della valutazione utenti	102

6.5 Considerazioni finali	102
<i>Ringraziamenti</i>	103
<i>Bibliografia</i>	105

ELENCO DELLE FIGURE

1.1	Corrispondenza tra obiettivi di ricerca e contributi del progetto SOULFRAME.	3
1.2	Struttura della tesi e organizzazione dei capitoli.	4
2.1	Evoluzione della maturità della ricerca sui Conversational Agents (CA), dalla “zero hour wave” alla “AI wave”, con riferimento ai principali passaggi tecnologici. ¹	7
2.2	Fattori immersivi associati alla presenza sociale, con qualità audio tra le variabili considerate in letteratura. ²	8
2.3	Schema della pipeline conversazionale (STT → RAG/LLM → TTS) adottata in SOULFRAME.	9
2.4	Architettura del modello Whisper: encoder Transformer con log-mel spectrogram in input e decoder per la trascrizione multilingue zero-shot. ³	10
2.5	Schema concettuale di Retrieval-Augmented Generation (RAG): recupero di contesto e generazione della risposta. ⁴	11
2.6	Panoramica dell’architettura XTTS: Conditioning Encoder, encoder GPT-2 con Perceiver Resampler e decoder HiFi-GAN per sintesi multilingue zero-shot con voice cloning. ⁵	12
3.1	Flusso end-to-end del sistema SOULFRAME: dall’input vocale dell’utente alla risposta dell’agente embodied.	19
3.2	Vista d’insieme dell’architettura a componenti di SOULFRAME: il client Unity WebGL comunica con i micro-servizi backend attraverso il reverse proxy Caddy.	20
3.3	Diagramma degli stati UI del client SOULFRAME: dalla fase di caricamento iniziale al ciclo conversazionale (Idle → Listening → Processing → Speaking) e alla gestione degli errori.	22

3.4	Flusso end-to-end di una richiesta conversazionale in SOULFRAME: dall'acquisizione audio push-to-talk alla riproduzione della risposta vocale.	23
3.5	Flusso di gestione avatar in SOULFRAME: dalla creazione tramite Avaturn alla cache server-side e al rendering nel client Unity. . . .	24
3.6	Interfaccia carosello avatar in SOULFRAME: selezione del profilo attivo tra avatar locali e importati.	25
3.7	Interfaccia <code>MainMenu</code> in SOULFRAME: accesso alle funzioni principali prima dell'onboarding dell'avatar.	26
3.8	Schermata <code>SetupVoice</code> : registrazione e validazione del campione vocale durante l'onboarding.	27
3.9	Schermata <code>SetupMemory</code> : inserimento note e ingestione contenuti per popolare la memoria per-avatar.	28
3.10	Stato di ascolto push-to-talk nel client Unity WebGL: acquisizione audio attiva con indicatore visivo.	29
3.11	MainMode conversazionale in SOULFRAME: trascrizione e risposta dell'avatar in corso durante un turno vocale.	30
3.12	Interfaccia Swagger UI (<code>/docs</code>) del servizio RAG di SOULFRAME: gli endpoint esposti da <code>rag_server.py</code> con schema automatico generato da FastAPI.	31
3.13	Struttura del filesystem di persistenza in SOULFRAME: tre store isolati per <code>avatar_id</code> (asset <code>.glb</code> , profilo vocale e memoria RAG).	32
4.1	Diagramma della macchina a stati <code>UIState</code> nel client SOULFRAME: i sei stati principali e le transizioni guidate da eventi utente e callback asincroni (bootstrap servizi, verifica profilo voce, verifica memoria).	34
4.2	Confronto tra UI desktop e UI touch in <code>MainMode</code> : a parità di <code>UIState</code> , cambiano input primario e <code>UIHintBar</code> (icone tastiera vs icone gesture/PTT).	36

4.3	Architettura dell'integrazione Avaturn in WebGL: AvaturnWebController invoca il bridge OpenAvaturnIframe in AvaturnBridge.jslib, che crea un overlay DOM e inizializza l'SDK Avaturn; l'evento di export ritorna a Unity tramite SendMessage con un payload JSON.	38
4.4	Stack di acquisizione audio in WebGL: AudioRecorder delega al provider IAudioCaptureWebGL (WebGLAudioCapture), che richiama il plugin AudioCapture.jslib per cattura MediaRecorder e con- versione WAV (header RIFF) con trasferimento dei byte via heap Emscripten.	41
4.5	Pipeline conversazionale in MainMode: PTT → WAV → Whisper/transcribe → RAG/chat → Coqui/tts_stream → PcmStreamDownloadHandler → PcmChunkPlayer → AudioSource.	49
4.6	Architettura dei micro-servizi backend: i quattro servizi FastA- PI (Whisper, RAG, Avatar Asset, TTS) comunicano su porte de- dicate; il servizio RAG si appoggia a Ollama (porta 11434) per embedding e chat LLM.	50
4.7	Ricerca ibrida nel servizio RAG: pesatura BM25/vettoriale (60/40), fusione e deduplicazione dei frammenti per il prompt LLM.	52
4.8	Protocollo di streaming audio TTS: header WAV iniziale, chunk PCM in tempo reale e riproduzione anticipata prima del completamento della sintesi.	56
4.9	Logica di normalizzazione degli endpoint in WebGL: se la pagina web è servita da loopback, si mantengono le URL assolute alle porte locali; altrimenti si usano path relativi per il routing tramite reverse proxy.	61
4.10	Ciclo di vita operativo dei servizi SOULFRAME gestito dagli script <code>sfctl</code> e <code>sf_admin_ubuntu.sh</code>	74
4.11	Livelli di sicurezza dell'architettura SOULFRAME: rete, reverse- proxy, applicazione, sistema operativo.	78
5.1	Interfaccia desktop in modalità MainMode con transcript, risposta dell'avatar e stato conversazionale in forma testuale.	88

5.2	Interfaccia touch in modalità MainMode: Push-to-Talk, swipe tra transcript e reply e adattamenti per l'usabilità su schermo touch.	89
5.3	Diagramma di sequenza di un failure case con timeout, retry e feedback visivo.	93

ELENCO DELLE TABELLE

3.1	Riepilogo dei micro-servizi backend di SOULFRAME: porte, endpoint principali e responsabilità operative.	31
3.2	Confronto operativo tra setup locale Windows e deploy server Ubuntu.	32
4.1	Endpoint backend effettivamente invocati dal client Unity e contratto minimo di input/output.	59
5.1	Confronto indicativo dei tempi per fase della pipeline tra ambiente locale e server. La riga TTS riporta il tempo al primo audio; la latenza end-to-end è n.d.	84
5.2	Tempo di inizializzazione (cold-start) osservato per rendere i servizi pronti all'interazione.	85

1. INTRODUZIONE

1.1 *Motivazione e contesto applicativo*

SOULFRAME nasce da un problema concreto: in molti ambienti VR la scena è credibile, ma la conversazione con l’agente resta fragile o troppo rigida. Questa tesi propone un prototipo in cui l’utente parla con un avatar embodied e riceve una risposta vocale contestuale nella stessa scena 3D.

La difficoltà è tenere insieme tre piani che spesso vengono trattati separatamente: interazione vocale a turni, continuità del dialogo e presenza dell’interlocutore virtuale. In VR non basta la qualità grafica: la percezione di presenza dipende anche dalla coerenza tra azioni, tempi di risposta ed eventi della scena [1]. Quando questi elementi si disallineano, l’agente appare poco affidabile anche se il contenuto linguistico è corretto.

Le evidenze sperimentali sugli ECA vocali indicano che STT+TTS in tempo reale può aumentare la co-presenza rispetto a soluzioni preregistrate [2]. La letteratura in XR mostra inoltre che la maggior parte dei prototipi è sviluppata in VR, spesso con Unity, e che cresce l’interesse per dialoghi più adattivi supportati da modelli neurali [3]. In SOULFRAME questa linea viene applicata a una configurazione senza HMD obbligatorio, orientata alla fruizione via browser.

L’idea chiave del progetto è trattare l’avatar come un profilo persistente composto da aspetto, voce e memoria. Così l’interazione non riparte da zero a ogni turno e mantiene una continuità minima tra sessioni. Nel Capitolo 2, Figura 2.1, è richiamata la traiettoria dei Conversational Agents dalla fase scriptata alle soluzioni recenti basate su LLM.

In questo contesto SOULFRAME non è una demo grafica isolata: combina pipeline vocale, memoria conversazionale e resa embodied per verificare quanto questi elementi incidano sulla qualità percepita dell’interazione.

1.2 Perimetro e requisiti di progetto

SOULFRAME è un prototipo di interazione immersiva con ECA vocale embodied in ambiente 3D. L’obiettivo è valutare fattibilità tecnica e qualità percepita in scenari con componente sociale. Non è una piattaforma generale per creare mondi virtuali e non è una soluzione clinica certificata.

Il perimetro resta concentrato sull’integrazione *end-to-end* tra voce, memoria conversazionale e rappresentazione embodied dell’agente. Ho scelto di vincolare l’accesso alla conversazione operativa a un onboarding minimo del profilo avatar, perché senza voce e memoria l’interazione diventa rapidamente *stateless*.

Sul piano funzionale, il nucleo è la pipeline a turni *push-to-talk*: acquisizione audio, trascrizione STT, generazione contestuale con RAG/LLM e risposta TTS. Funzioni come descrizione immagini e OCR rientrano nel prototipo come capacità di supporto alla memoria; i dettagli implementativi sono discussi nei capitoli successivi.

I requisiti non funzionali riguardano soprattutto latenza percepita, robustezza ai guasti parziali e modularità dei servizi. La tesi adotta quindi criteri operativi verificabili sul prototipo, evitando soglie astratte non allineate al contesto d’uso reale.

1.3 Obiettivi e contributi di SOULFRAME

Gli obiettivi di SOULFRAME si collocano nel filone degli ECA in XR, dove sono comuni implementazioni VR in Unity e interazioni vocali, ma con metodi di valutazione ancora eterogenei [3]. Il progetto punta a costruire un prototipo in cui l’integrazione *end-to-end* sia osservabile e discutibile con criteri esplicativi.

RQ1 riguarda la fattibilità tecnica di un’interazione vocale in tempo reale con un agente embodied in ambiente immersivo 3D. In termini operativi, la catena STT → RAG/LLM → TTS → output audiovisivo deve funzionare in modo continuo, includendo anche le capacità di supporto (descrizione immagini e OCR documentale). La valutazione considera tempi per blocco, completamento dei turni senza errori bloccanti e coerenza delle transizioni tra ascolto, elaborazione e risposta.

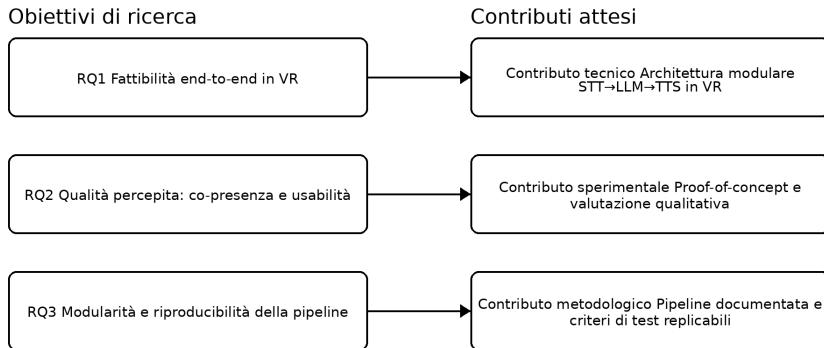


Fig. 1.1: Corrispondenza tra obiettivi di ricerca e contributi del progetto SOULFRAME.

RQ2 riguarda la qualità percepita dell’interazione, con attenzione a co-presenza e naturalezza dialogica. SOULFRAME prevede una valutazione qualitativa guidata, affiancata quando possibile da metriche soggettive usate negli studi VR con ECA. Studi comparativi tra agenti conversazionali e audio pre-scriptato mostrano differenze osservabili, soprattutto sulla co-presenza [2].

RQ3 riguarda modularità e riproducibilità della pipeline. Il sistema deve permettere la sostituzione dei componenti principali senza riscrivere l’intera architettura e rendere tracciabili configurazioni, tempi e risultati delle prove. Figura 1.1 sintetizza la corrispondenza tra i tre obiettivi e i contributi attesi.

Sul piano tecnico, il progetto introduce un’architettura modulare *end-to-end* basata su micro-servizi (STT, RAG/LLM, TTS e memoria) con interfacce API esplicite. Sul piano sperimentale, il risultato è un *proof-of-concept* verificabile con un set minimo di misure soggettive e osservazioni qualitative. Per la replicabilità, resta centrale la documentazione delle scelte progettuali, in linea con il dibattito sul *dialogue management* e sulla necessità di metriche più comparabili [4].

1.4 Panoramica del sistema e flusso end-to-end

SOULFRAME adotta un’architettura client–server per sostenere un dialogo vocale a turni in ambiente immersivo 3D. Il client Unity WebGL gestisce scena, avatar e interfaccia; il backend gestisce trascrizione, memoria conversazionale, generazione testuale e sintesi vocale.

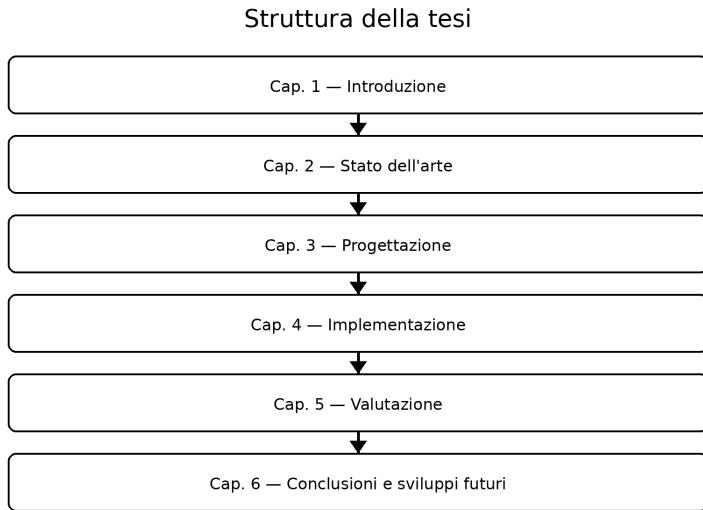


Fig. 1.2: Struttura della tesi e organizzazione dei capitoli.

Il flusso è lineare a livello concettuale: input vocale dell’utente, trascrizione STT, generazione contestuale con RAG/LLM, risposta TTS e playback sull’avatar. Le stesse fasi devono però restare coordinate nei tempi, perché ritardi o transizioni incoerenti compromettono la naturalezza percepita.

Ho scelto una fruizione via browser WebGL senza HMD obbligatorio per abbassare la barriera di accesso durante sviluppo e test. In parallelo ho mantenuto una pipeline modulare a micro-servizi, così da poter sostituire i blocchi principali senza riscrivere il client.

La vista d’insieme architetturale è riportata nel Capitolo 3, in Figura 3.1, dove la pipeline è discussa con maggior dettaglio tecnico.

1.5 Metodo di lavoro e struttura della tesi

Lo sviluppo di SOULFRAME segue un approccio iterativo basato su prototipazione incrementale. Questa scelta riduce il rischio tecnico tipico dei sistemi che combinano reattività a turni (*push-to-talk*), elaborazione linguistica e rendering immersivo. Il lavoro procede per integrazioni successive: prima validazione dei moduli in isolamento, poi integrazione della pipeline completa e verifica su scenari progressivamente più complessi. Durante il processo vengono traccia-

ti compromessi e dipendenze per mantenere l’evoluzione del sistema leggibile e replicabile.

La tesi segue una progressione lineare: dai fondamenti teorici si passa all’architettura, poi alle scelte implementative e infine alla valutazione sperimentale del prototipo. L’ultimo passaggio raccoglie limiti emersi e sviluppi futuri, mantenendo continuità tra decisioni progettuali e risultati osservati. La sequenza è stata pensata per separare chiaramente il “perché” delle scelte dal “come” tecnico con cui sono state realizzate.

Figura 1.2 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 per 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. Sono frequenti anche l'interazione vocale e le 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.

La traiettoria che porta dai CA scriptati ai sistemi basati su modelli linguistici è sintetizzata in Figura 2.1. Il passaggio chiarisce perché embodiment e gestione del dialogo vengano oggi trattati insieme nei prototipi XR.

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. In valutazione conviene scomporre questo fenomeno in componenti distinguibili, includendo la co-presenza (percezione di condividere lo stesso spazio) e

¹ Fonte: adattamento da una figura pubblicata su ResearchGate.

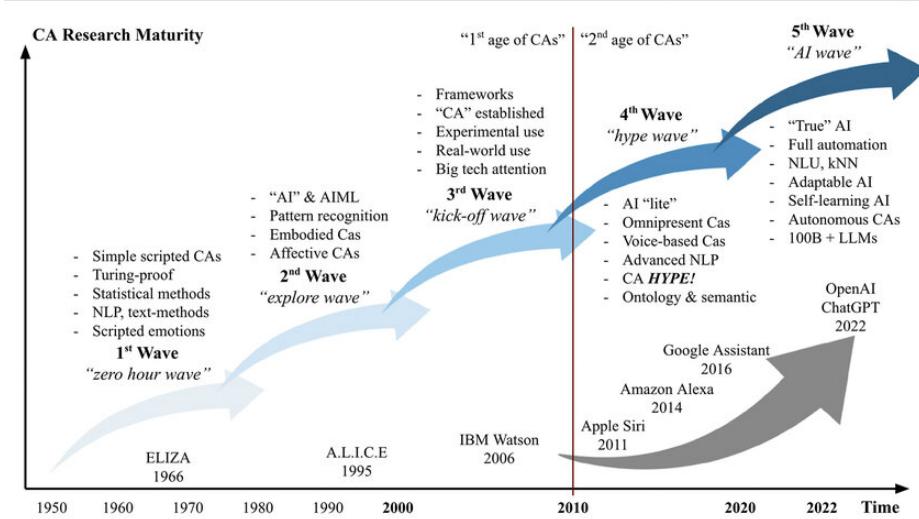


Fig. 2.1: Evoluzione della maturità della ricerca sui Conversational Agents (CA), dalla “zero hour wave” alla “AI wave”, con riferimento ai principali passaggi tecnologici.¹

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 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. In Figura 2.2 la qualità audio compare tra le variabili immersive considerate in letteratura. 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 mitigano sovrapposizioni e silenzi durante l'elaborazione.

² 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).

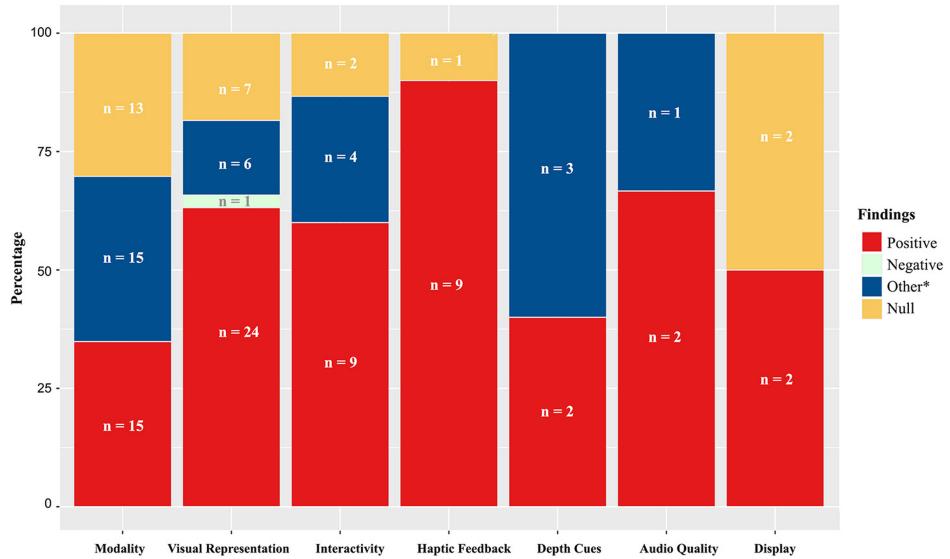


Fig. 2.2: Fattori immersivi associati alla presenza sociale, con qualità audio tra le variabili considerate in letteratura.²

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.

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.

Pipeline AI a tre stadi



Fig. 2.3: Schema della pipeline conversazionale (STT → RAG/LLM → TTS) adottata in SOULFRAME.

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 isola i vincoli della conversazione a turni, soprattutto la latenza, e permette di aggiornare i singoli moduli senza modificare l'intera architettura. Figura 2.3 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

In SOULFRAME, Whisper è scelto come base STT per la robustezza zero-shot in condizioni non controllate, evitando l'onere di addestrare un ASR dedicato al progetto. Il primo stadio della pipeline è il riconoscimento automatico del parlato: l'audio acquisito in push-to-talk viene trascritto e inoltrato ai moduli conversazionali successivi.

Whisper è un 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] In questa sede interessa soprattutto il ruolo teorico del modello nella pipeline; configurazioni operative e profili di deploy sono discussi nei capitoli architetturali e implementativi.

³ Fonte: Radford et al., *Robust Speech Recognition via Large-Scale Weak Supervision*, ICML 2023, Fig. 1, <https://arxiv.org/abs/2212.04356>.

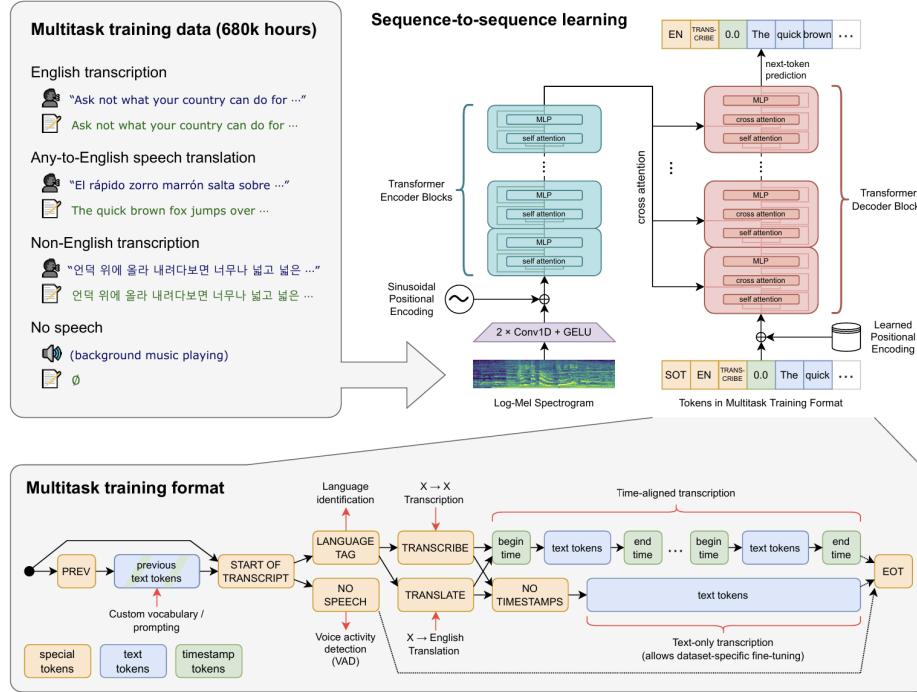


Fig. 2.4: Architettura del modello Whisper: encoder Transformer con log-mel spectrogram in input e decoder per la trascrizione multilingue zero-shot.³

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

In SOULFRAME, il paradigma RAG è scelto per aggiornare la memoria conversazionale senza riaddestrare il modello generativo a ogni variazione dei contenuti. Il secondo stadio introduce quindi una memoria che combina generazione e recupero di contesto.

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). Così 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 2.5 mostra il flusso retrieval→generation.

⁴ Fonte: Kaur et al., *Knowledge, context and personalization in retrieval-augmented generation for healthcare*, Frontiers in Artificial Intelligence (2025), <https://www.frontiersin.org/>

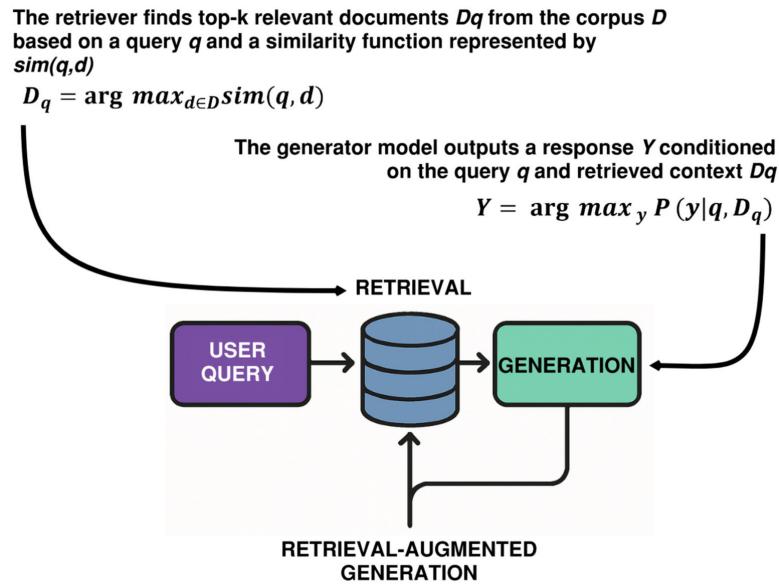


Fig. 2.5: Schema concettuale di Retrieval-Augmented Generation (RAG): recupero di contesto e generazione della risposta.⁴

Nel quadro della tesi, il punto rilevante è che RAG separa la conoscenza dinamica dal solo comportamento parametrico del modello, rendendo praticabile una memoria per-avatar che può essere estesa nel tempo. Le scelte operative su runtime, storage vettoriale e retrieval sono rinviate ai capitoli di architettura e implementazione.

2.2.3 Text-to-Speech con Coqui XTTS v2

In SOULFRAME, XTTS v2 è scelto per ottenere voce per-avatar in modalità zero-shot, evitando pipeline basate su risposte preregistrate statiche. Il terzo stadio converte il testo in audio e chiude il ciclo conversazionale: in un ECA embodied la sintesi influisce direttamente sulla percezione di continuità e coerenza dell'interlocutore.

Coqui XTTS v2 offre supporto multilingue e voice cloning tramite campione di riferimento, senza richiedere l'addestramento di un modello vocale dedicato per ogni utente.[9] In questo capitolo il focus resta sulla rilevanza teorica del modello per il problema di ricerca; i dettagli di configurazione e integrazione sono trattati

journals/artificial-intelligence/articles/10.3389/frai.2025.1697169/full (Figura 1; licenza/uso: CC BY).

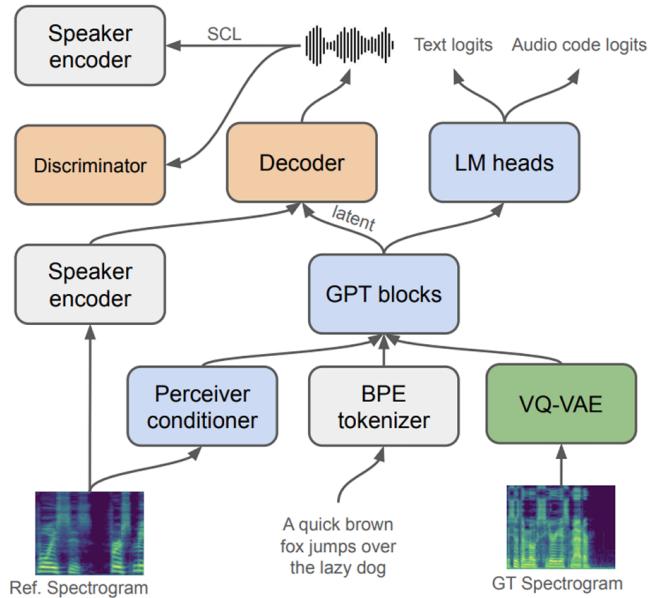


Fig. 2.6: Panoramica dell’architettura XTTS: Conditioning Encoder, encoder GPT-2 con Perceiver Resampler e decoder HiFi-GAN per sintesi multilingue zero-shot con voice cloning.⁵

nei capitoli successivi. Un riferimento architettonicale del modello è riportato in Figura 2.6.

2.3 Posizionamento di SOULFRAME rispetto allo stato dell’arte

La scelta di una pipeline locale a tre stadi nasce soprattutto dal problema della latenza percepita: ridurre dipendenze cloud aiuta a mantenere più prevedibile il ciclo audio→testo→risposta→audio. In SOULFRAME, la continuità è sostenuta da una memoria persistente e isolata per avatar, che conserva contesto e preferenze senza interferenze tra profili distinti. Nel retrieval, la combinazione ibrida di segnali semanticci e lessicali rende la risposta più robusta anche con query formulate in modo variabile. Per contenere la latenza percepita, il sistema usa sintesi in streaming e strategie di attesa che riducono i silenzi udibili. La modularità architettonicale permette di sostituire singoli moduli senza modificare il client, favorendo iterazione tecnica e confrontabilità dei risultati.

⁵ Fonte: Casanova et al., *XTTS: a Massively Multilingual Zero-Shot Text-to-Speech Model*, Interspeech 2024, Fig. 1, <https://arxiv.org/abs/2406.04904>.

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. La memoria non è limitata a note manuali: ingestione di documenti e immagini trasformano materiali esterni in contesto riutilizzabile e ampliano il perimetro oltre il dialogo puro. L'esecuzione via browser riduce la barriera di ingresso rispetto a configurazioni VR tradizionali, ma impone vincoli tecnici su microfono, gestione audio e latenza. In questo quadro diventano centrali 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

In SOULFRAME 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/>

² Avaturn, *Avaturn Integration Documentation*, <https://docs.avaturn.me/docs/integration/overview/>

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. Per questo conviene 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 12 s in esecuzione locale e entro 16 s in deploy pubblico. Il target è calibrato sulle misure osservate di STT+TTS e include un margine operativo per la fase RAG/LLM, non misurata sistematicamente per singolo stadio. Questa metrica non coincide con il tempo di cold-start iniziale dei servizi. La conformità al requisito è verificata tramite timestamp lato client sui turni vocali, come nella metodologia di misura riportata nel Capitolo 5, 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. 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).

RNF4 (Coerenza estetica e comunicazione di stato): il linguaggio visivo deve

restare coerente con la natura stilizzata degli avatar Avaturn e con il carattere sperimentale del prototipo. Nel client questo vincolo si traduce in effetti di post-processing, rings animati e feedback di selezione, scelti perché compatibili con rendering WebGL e utili a rendere leggibili gli stati operativi senza aumentare il carico cognitivo dell’utente. Il requisito è soddisfatto se gli indicatori visivi accompagnano in modo consistente transizioni e operazioni principali dell’interfaccia (inizializzazione, setup, onboarding, selezione), mantenendo continuità tra resa dell’avatar e contesto UI.

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. In locale Windows il profilo predefinito è `small`; nel setup Ubuntu testato il profilo usato è `medium`, grazie a una VRAM disponibile maggiore (circa 24 GB contro circa 12 GB in locale). 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 varia tra ambiente locale Windows e deploy Ubuntu: in locale è adottata la variante `llama3:8b-instruct-q4_K_M`, mentre sul server Ubuntu la configurazione di default è `llama3.1:8b`; il modello embedding resta invece 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 endpoint e formati attesi. I dettagli implementativi di questa parametrizzazione sono discussi in sottosezione 4.2.2.

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 progetto Caddy è stato preferito a Nginx per tre motivi operativi: TLS automatico, redirect HTTP→HTTPS integrato e configurazione più compatta in fase di bootstrap. Nel prototipo, gli endpoint non implementano autenticazione/autorizzazione applicativa e l'architettura assume una rete di esecuzione controllata. Figura 3.2 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.

Per una vista sintetica della catena conversazionale completa, Figura 3.1 mostra il flusso end-to-end dall'input vocale alla risposta dell'agente embodied.

Operativamente, 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)

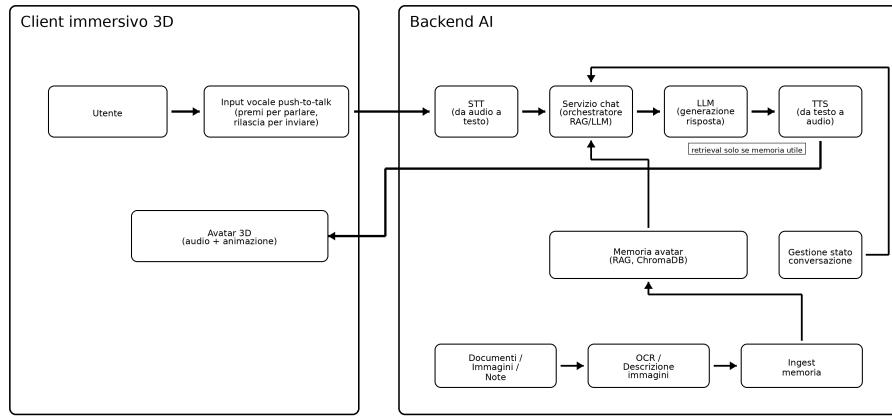


Fig. 3.1: Flusso end-to-end del sistema SOULFRAME: dall’input vocale dell’utente alla risposta dell’agente embodied.

per ridurre conflitti tra processi durante lo sviluppo. In ambiente Ubuntu, ciascun micro-servizio è gestito come unità `systemd` (`systemd`: gestore di processi e servizi del sistema operativo Linux, responsabile dell’avvio automatico e del restart dei servizi), con comandi amministrativi che permettono `start/stop/status` e aggiornamenti senza dover riconfigurare manualmente l’intero stack. Questa separazione è rimasta invariata durante lo sviluppo 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 $\text{audio} \rightarrow \text{testo} \rightarrow \text{risposta} \rightarrow \text{audio}$

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

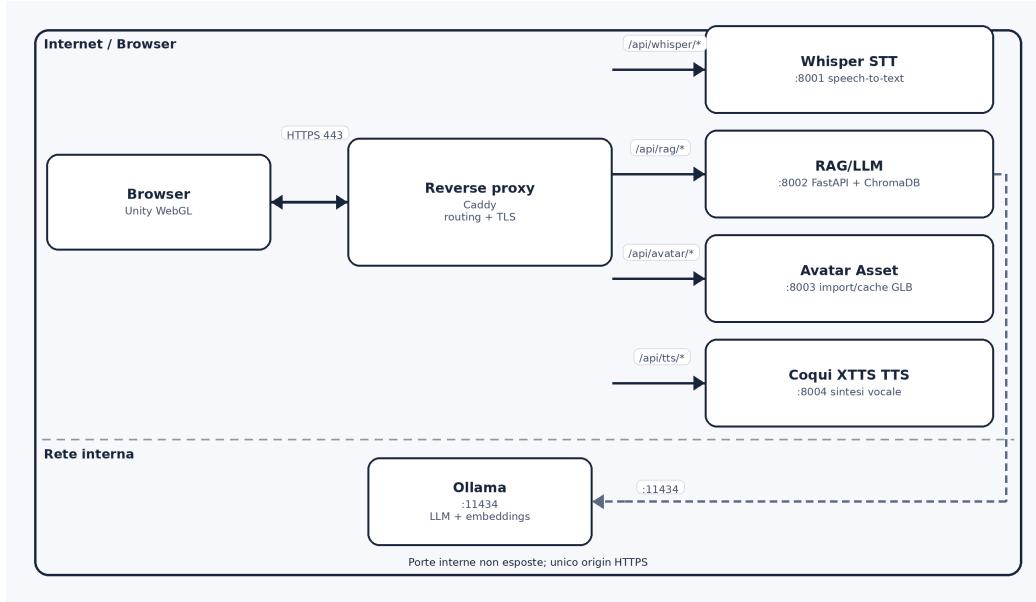


Fig. 3.2: Vista d'insieme dell'architettura a componenti di SOULFRAME: il client Unity WebGL comunica con i micro-servizi backend attraverso il reverse proxy Caddy.

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.

Il servizio RAG/LLM riceve quindi la trascrizione tramite 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 configurazione corrente `nomic-embed-text` erogato da Ollama). La scelta di ChromaDB privile-

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

⁴ Chroma, *Chroma Documentation*, <https://docs.trychroma.com/>

gia leggerezza e semplicità di esecuzione locale (senza servizi esterni o account) e si integra in modo diretto con la gerarchia per-avatar `rag_store/<avatar_id>`; alternative come FAISS possono offrire throughput maggiore, ma richiedono una gestione più artigianale della persistenza e del layout per profilo nel contesto di questo prototipo. 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 adotta meccanismi di warmup durante il boot; 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 usa un fallback audio di cortesia per evitare silenzi prolungati e dare un segnale immediato di reattività. I dettagli implementativi di warmup, wait phrases e streaming sono discussi nel Capitolo 4.

Figura 3.4 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

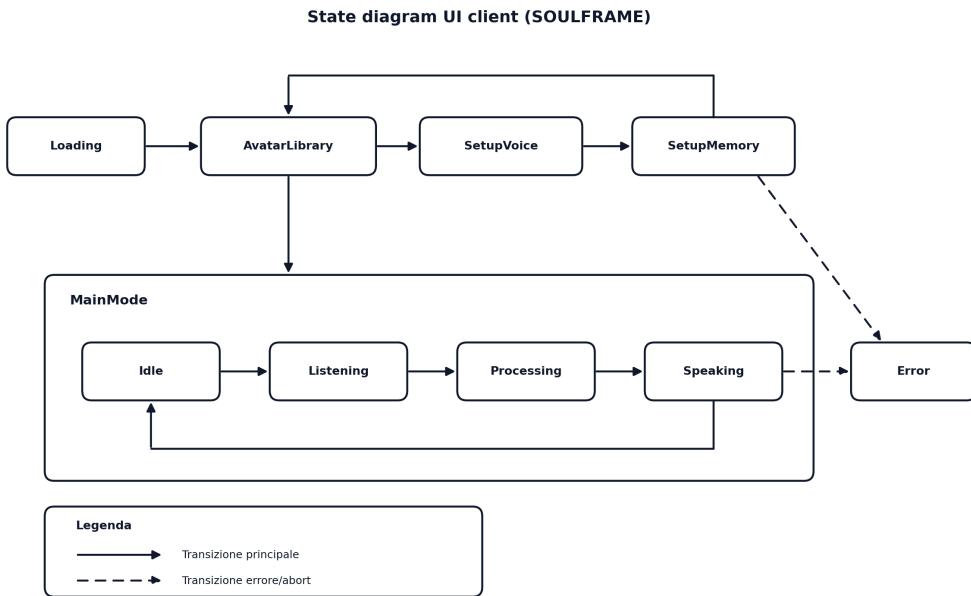


Fig. 3.3: Diagramma degli stati UI del client SOULFRAME: dalla fase di caricamento iniziale al ciclo conversazionale (Idle → Listening → Processing → Speaking) e alla gestione degli errori.

tecniche di riduzione della latenza percepita. La stessa figura chiarisce che, in deploy con reverse proxy, gli endpoint REST restano invariati a livello logico e vengono raggiunti tramite path riscritti sotto un unico origin HTTPS.

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

Il flusso di gestione avatar in SOULFRAME affianca due percorsi che convergono nella stessa esperienza in scena: avatar locali pre-inclusi nella build e avatar importati dall’utente. L’obiettivo architettonale è garantire sempre una libreria minima selezionabile, mantenendo allo stesso tempo un canale controllato per acquisire e servire asset esterni in formato .glb.

Nel percorso locale, il client seleziona un profilo già disponibile e lo carica direttamente in scena. Nel percorso importato, il profilo viene creato esternamente, importato dal backend e poi reso disponibile nella stessa libreria del client. In entrambi i casi, il punto di convergenza resta identico: selezione del profilo, caricamento del modello e attivazione dell’avatar nella pipeline conversazionale.

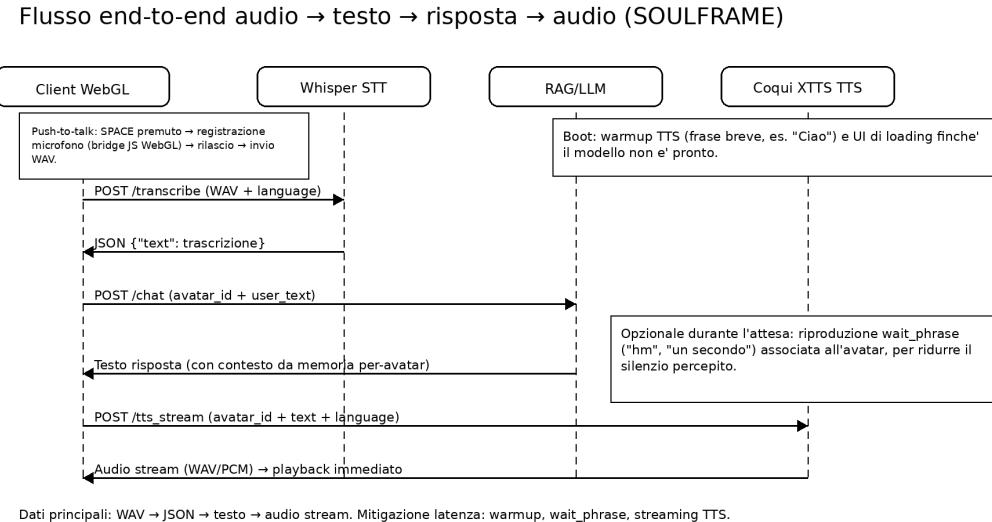


Fig. 3.4: Flusso end-to-end di una richiesta conversazionale in SOULFRAME: dall’acquisizione audio push-to-talk alla riproduzione della risposta vocale.

Dopo selezione o creazione, il flusso prosegue con onboarding voce e memoria per-avatar. L’accesso a MainMode avviene solo quando i prerequisiti minimi del profilo risultano soddisfatti, così da evitare conversazioni prive di contesto operativo. I dettagli implementativi del bridge WebGL, del ciclo di import e delle logiche di robustezza lato asset sono trattati nel Capitolo 4.

3.3 Componenti implementati

I componenti implementati rendono operativa l’architettura della Sezione 3.2: integrazione Avaturn nel client Unity WebGL, backend AI a micro-servizi e persistenza per profilo avatar. La panoramica unificata dei servizi backend è riportata in Tabella 3.1; i dettagli fini di funzioni e ottimizzazioni sono nel Capitolo 4.

3.3.1 Integrazione Avaturn nel frontend Unity WebGL

L’integrazione Avaturn nel frontend Unity WebGL segue una logica a due passaggi: creazione del profilo avatar in ambiente web dedicato e successivo import del modello .glb nel flusso applicativo di SOULFRAME. A livello architettonico conta il punto di convergenza: una volta completata la creazione, il client

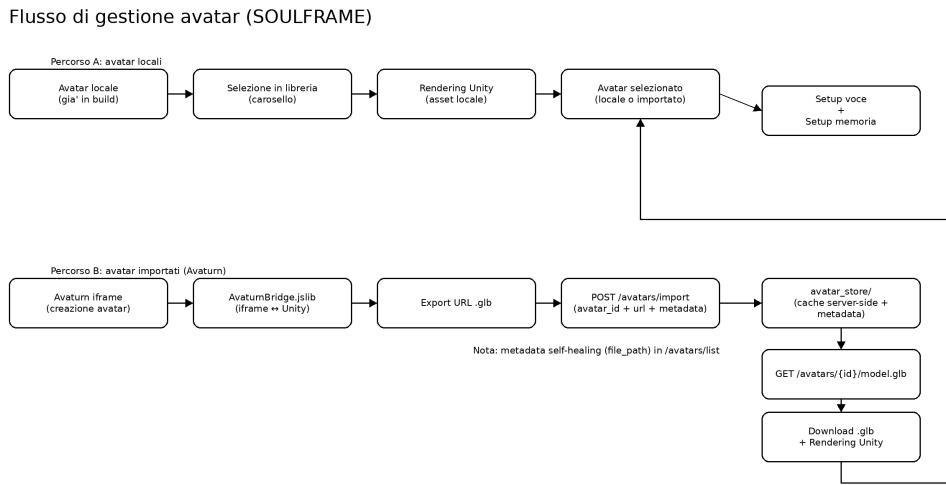


Fig. 3.5: Flusso di gestione avatar in SOULFRAME: dalla creazione tramite Avaturn alla cache server-side e al rendering nel client Unity.

acquisisce l’identità dell’avatar e delega l’import al backend, riallineandosi poi al percorso unico descritto in Sezione 3.2. Nel runtime Unity il caricamento dei .glb avviene con GLTFast⁵. I dettagli implementativi del bridge WebGL, della gestione iframe/DOM e dell’interoperabilità C#/JavaScript sono trattati nel Capitolo 4 (sottosezione 4.1.3).

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 permette 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, collegando porte, endpoint e responsabilità operative in un’unica vista.

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

⁵ Unity Technologies, *glTFast package documentation*, <https://docs.unity3d.com/Package/com.unity.cloud.gltfast@6.10/manual/index.html>



Fig. 3.6: Interfaccia carosello avatar in SOULFRAME: selezione del profilo attivo tra avatar locali e importati.

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; nel prototipo il bilanciamento 0.6/0.4 favorisce leggermente BM25 perché le query utente contengono spesso keyword e nomi propri, mantenendo comunque una quota semantica utile per recuperare ricordi parafrasati. In scenari diversi restano praticabili varianti 0.5/0.5 (embedding più stabili) o 0.7/0.3 (maggiore controllo sul semantic drift).

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. Questo vincolo consente di 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. In Figura 3.12 è riportata la documentazione automatica degli endpoint.

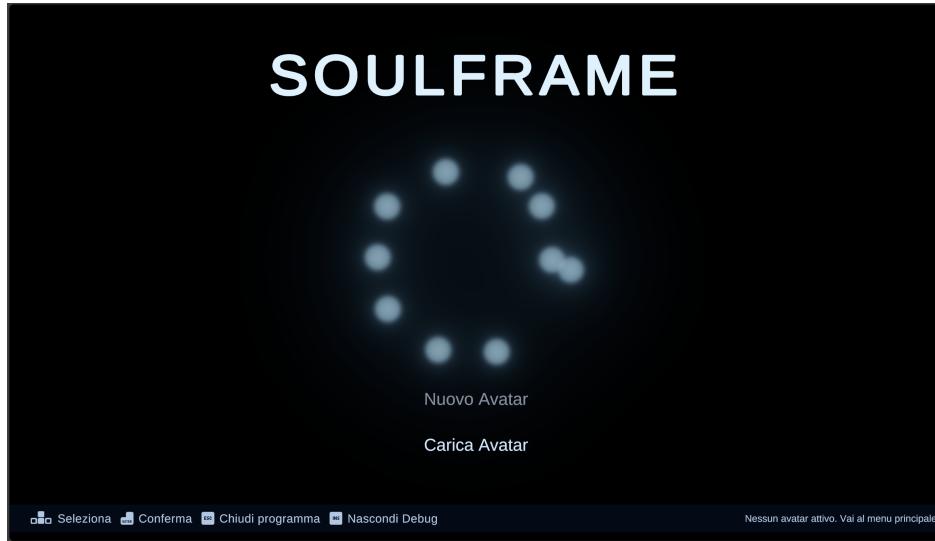


Fig. 3.7: Interfaccia MainMenu in SOULFRAME: accesso alle funzioni principali prima dell'onboarding dell'avatar.

3.3.3 Persistenza e gestione dati per avatar

Nel sistema, la persistenza dei dati in SOULFRAME è organizzata su file-system 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 interferenze 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 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



Fig. 3.8: Schermata SetupVoice: registrazione e validazione del campione vocale durante l'onboarding.

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

In ambiente locale la configurazione 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.

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.

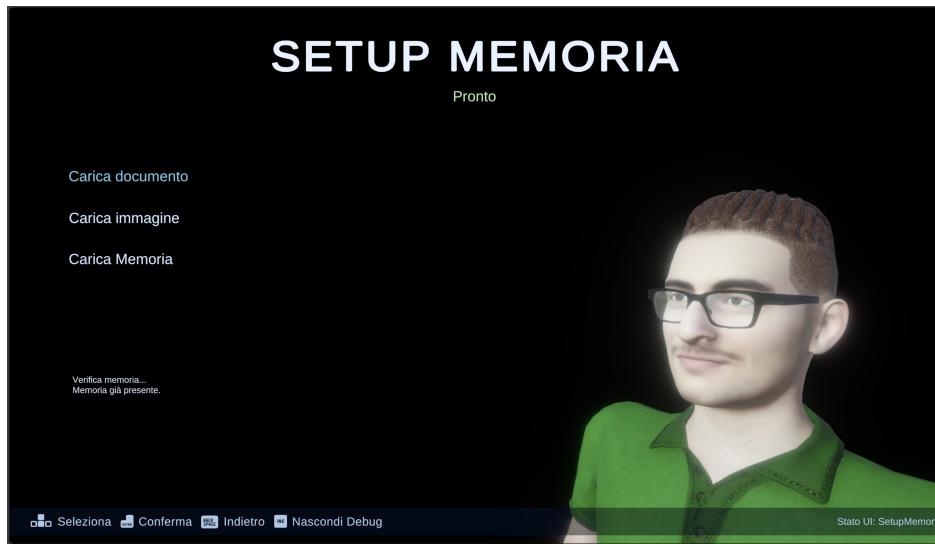


Fig. 3.9: Schermata SetupMemory: inserimento note e ingestione contenuti per popolare la memoria per-avatar.

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. Questa scelta 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 dedicati, 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.

Nel routing, il reverse proxy gestisce terminazione TLS, serving dei file statici e inoltro delle richieste API verso i servizi interni. La stessa build Unity Web-



Fig. 3.10: Stato di ascolto push-to-talk nel client Unity WebGL: acquisizione audio attiva con indicatore visivo.

GL resta così riutilizzabile tra locale e server, variando soprattutto il livello di orchestrazione e il punto di ingresso di rete. 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 mantiene 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



Fig. 3.11: MainMode conversazionale in SOULFRAME: trascrizione e risposta dell'avatar in corso durante un turno vocale.

verificabili queste differenze tra ambienti; i dettagli esecutivi sono approfonditi nel Capitolo 4.

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 .gbl, 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.

Fig. 3.12: Interfaccia Swagger UI (/docs) del servizio RAG di SOULFRAME: gli endpoint esposti da `rag_server.py` con schema automatico generato da FastAPI.

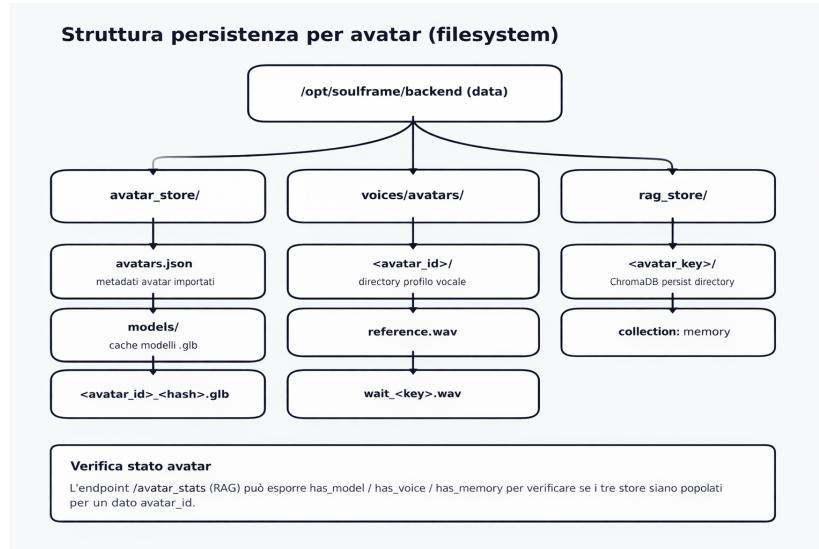


Fig. 3.13: Struttura del filesystem di persistenza in SOULFRAME: tre store isolati per `avatar_id` (asset .glb, profilo vocale e memoria RAG).

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.

4. SVILUPPO DEL PROGETTO: IMPLEMENTAZIONE E CRITICITÀ

4.1 *Implementazione frontend*

Questo capitolo descrive le scelte implementative principali e i problemi concreti incontrati durante lo sviluppo. Per ogni componente si indica cosa fa, perché è stato strutturato in quel modo e dove sono emerse le difficoltà più rilevanti. Unity è stato adottato perché offre un buon compromesso tra integrazione UI/3D/audio in WebGL e rapidità di iterazione; sul piano pratico, era anche lo strumento già consolidato durante il percorso dell'esame di Realtà Virtuale.

4.1.1 *Gestione stati UI e navigazione*

Nel client Unity WebGL di SOULFRAME la navigazione è governata da una macchina a stati finiti implementata in `UIFlowController.cs`. La FSM esplicita risolve un problema pratico: le stesse schermate devono restare coerenti mentre convivono input eterogenei, chiamate asincrone ai servizi e transizioni visuali; senza uno stato centrale, i mismatch tra pannello attivo, controlli disponibili e feedback UI diventano difficili da riprodurre e correggere.

La struttura usa `UIState {Boot, MainMenu, AvatarLibrary, SetupVoice, SetupMemory, MainMode}`, una mappa `panelMap` e una pila `backStack`. In `BuildPanelMap()` la corrispondenza è esplicita (`pnlMainMenu, pnlAvatarLibrary, pnlSetupVoice, pnlSetupMemory, pnlMainMode`), così ogni transizione parte da un riferimento univoco.

La parte visiva del cambio stato è centralizzata in `TransitionPanels(...)`. Ogni pannello entra con uno slide-in orizzontale: `transitionDuration` e `slideOffset` governano l'interpolazione, `ApplySlideOffset(...)` sposta il `RectTransform`, mentre `CanvasGroup` gestisce `alpha`, `interactable` e

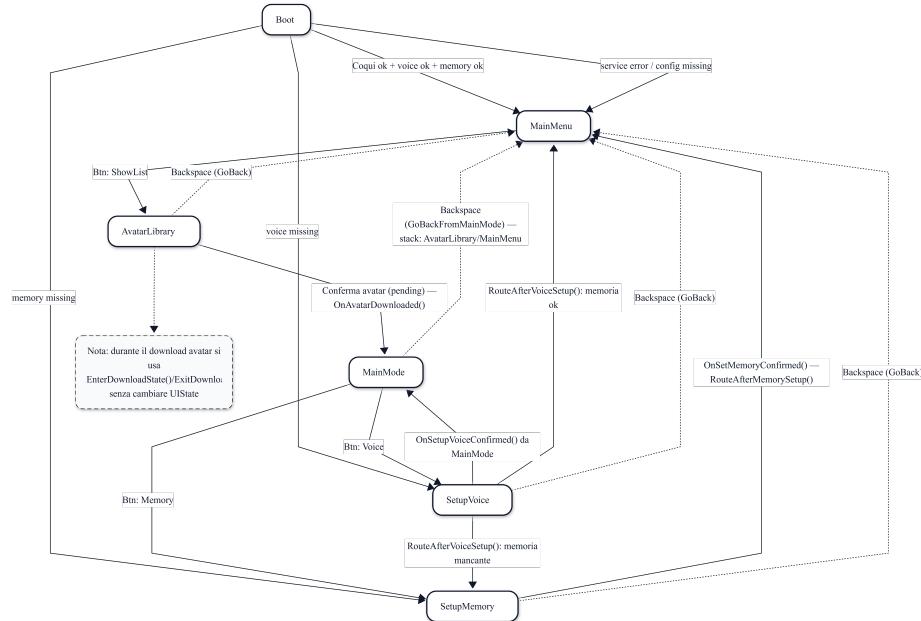


Fig. 4.1: Diagramma della macchina a stati UIState nel client SOULFRAME: i sei stati principali e le transizioni guidate da eventi utente e callback asincroni (bootstrap servizi, verifica profilo voce, verifica memoria).

blocksRaycasts. A fine transizione `ResetPanelPosition(...)` riporta il pannello alla posizione base. Questo schema evita salti visivi e soprattutto blocca click/focus residui durante i passaggi di stato.

Figura 4.1 riassume la FSM principale, mettendo in evidenza che `Boot` è un nodo decisionale: prima verifica disponibilità servizi e prerequisiti del profilo avatar, poi instrada verso onboarding o menu. Questo passaggio anticipato evita che errori strutturali emergano nel mezzo della conversazione, dove il recupero sarebbe più costoso in termini di UX.

Nei singoli stati ho mantenuto solo i casi non ovvi. In `AvatarLibrary` il carosello 3D non usa `Selectable` standard: la navigazione a focus resta quindi limitata e la selezione è delegata a `AvatarLibraryCarousel`, per evitare conflitti tra logica UI tradizionale e interazione con oggetti 3D. In `SetupMemory`, invece, la scelta chiave è separare input testuale e navigazione: quando l'input nota è in focus, la grammatica dei tasti cambia per prevenire collisioni tra editing e comandi globali.

`UINavigator.cs` e `UIHintBar.cs` completano questo schema. Restano distinti

dalla FSM: il primo traduce input discreti in azioni consistenti con lo stato, la seconda rende esplicita la grammatica corrente e riduce tentativi/errore.

In desktop, la hint bar usa set di icone dinamici aggiornati per stato. `UpdateHintBar(...)` alterna frecce verticali/orizzontali con `hintBar.SetArrowsHorizontal(...)` e, quando il PTT è attivo, commuta l'icona spazio con `hintBar.SetSpacePressed(...)`; in `UIHintBar.cs` questo passaggio usa `spaceAsset`/`spaceOutlinedAsset`. Nello stesso ciclo compaiono token contestuali come `Backspace`, `Delete`, `Any` e `INS`. In touch, `UpdateTouchHintBar(...)` imposta `SetTouchHints(...)` con sprite `Tap`, `Hold/HoldActive` e `SwipeHorizontal`; in parallelo `SetTouchPttVisualState()` alterna `MicIdle`/`MicActive` sui pulsanti PTT. Il razionale UX è pratico: ridurre ambiguità e mostrare subito quale azione è disponibile in quel momento.

Nello stato `AvatarLibrary` lo stesso principio viene esteso alle operazioni di rimozione profilo. In `DeleteSelectedAvatarRoutine()` il flusso distingue avatar locali e importati: per i locali applica un reset, per gli importati esegue anche la rimozione remota e l'aggiornamento della libreria. La conferma visiva nel carosello è coerente con questa differenza: `AvatarLibraryCarousel.cs` usa `PlayResetEffect()`/`ResetEffectRoutine(...)` per una rotazione completa con piccolo salto, mentre `PlayDeleteEffect()` /`DeleteEffectRoutine(...)` combina rotazione e discesa progressiva fino alla scomparsa dell'elemento. Anche il feedback audio resta allineato all'azione: `UINavigator.cs` espone `resetClip`, `deleteClip` ed `errorClip`, richiamati da `UIFlowController` con `navigator.PlayResetClip()`, `navigator.PlayDeleteClip()` e, nei fallimenti, tramite `PlayErrorClip()` che inoltra a `navigator.PlayErrorClip()`. In pratica, l'utente riceve una conferma multimediale immediata (testo, animazione, suono), con minore ambiguità sull'esito dell'azione e minore carico cognitivo nei passaggi critici.

La modalità touch non replica la desktop: riusa lo stesso flusso stato per stato, ma cambia affordance e priorità input (tap/hold al posto del focus). La separazione tra comportamento applicativo e presentazione limita le divergenze al livello di binding input. Il confronto visivo in Figura 4.2 mostra proprio questa strategia: cambia il linguaggio d'interazione, non il ciclo logico sottostante.

In WebGL la robustezza dipende anche dalla gestione dei contesti esterni:

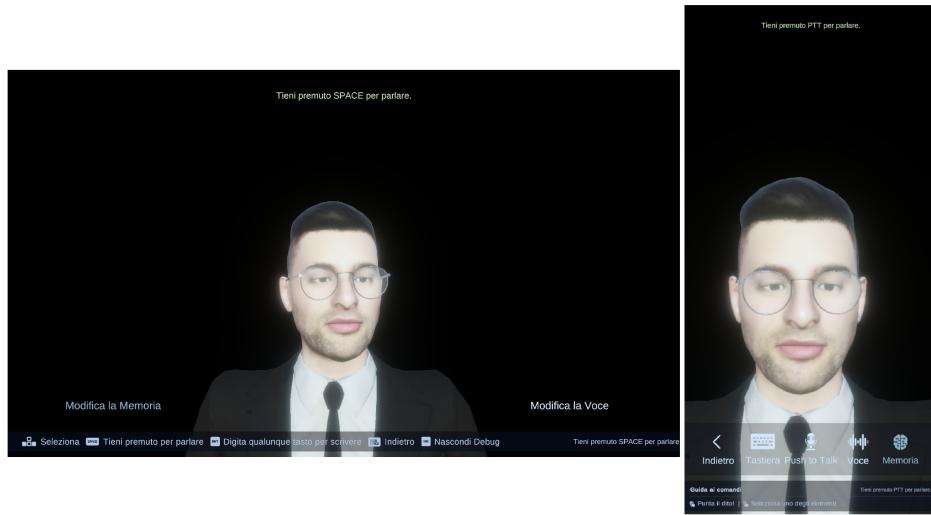


Fig. 4.2: Confronto tra UI desktop e UI touch in MainMode: a parità di UIState, cambiano input primario e UIHintBar (icone tastiera vs icone gesture/PTT).

quando è aperto un overlay browser o la UI è in lock esplicito, il navigatore sospende l'input e l'`EventSystem` viene tenuto separato dalla hint bar. La separazione tra input e hint bar evita che eventi tastiera o focus residui alterino lo stato corrente in modo non intenzionale, problema frequente quando Unity condivide la pagina con componenti DOM.

4.1.2 Gestione avatar e onboarding con Avaturn

In SOULFRAME la gestione avatar è concentrata in `AvatarManager`: non come semplice loader 3D, ma come coordinatore tra UI, persistenza e backend asset durante onboarding Avaturn¹. Questo accentramento nasce da un vincolo pratico: nel flusso reale import, download, cancellazione e selezione avvengono in tempi diversi rispetto all'input utente; separare la logica tra più componenti avrebbe aumentato race condition e stati intermedi difficili da validare.

Il profilo è serializzato come `AvatarData`. Il formato conserva solo i riferimenti minimi necessari a ricostruire l'asset e delega a runtime la risoluzione della sorgente effettiva in base a piattaforma e disponibilità locale. URL hardcoded non funzionano quando la stessa build passa da ambiente locale a reverse proxy; questa risoluzione a runtime mantiene il deploy stabile senza ramificare il codice.

¹ <https://docs.avaturn.me/docs/integration/overview/>

Sul desktop la persistenza locale (`Avatars.json`) resta utile, ma in WebGL viene ridotta intenzionalmente e la lista viene trattata come responsabilità backend, così cache browser e stato server non divergono nel tempo. Per lo stesso motivo la richiesta a `/avatars/list` resta il percorso principale anche fuori da WebGL: il fallback locale è un meccanismo di resilienza, non la fonte canonica.

Per la risoluzione URL, `AvatarManager` e `SoulframeServicesConfig` (`ScriptableObject`²) affrontano un vincolo operativo ricorrente: lo stesso avatar può essere referenziato con path locali, URL assolute o endpoint riscritti dal proxy. La normalizzazione resta vicina al punto d'uso per ridurre rotture in produzione dovute a differenze di host e routing.

Preview e caricamento principale sono separati per responsabilità, non per formato: la preview deve mantenere reattivo il carosello, il caricamento principale deve produrre un avatar pronto alla conversazione. Per questo motivo il flusso usa session identifier distinti, flag dedicati e cancellazioni indipendenti, così un risultato tardivo non può sovrascrivere lo stato corrente quando l'utente ha già cambiato contesto.

Nel percorso WebGL, `/avatars/import` e `WebGLDownloadAndInstantiate` costruiscono un pipeline asincrona che termina con istanziazione runtime via GLTFast³. Il caricamento monolitico non consente interruzioni pulite: l'assemblaggio runtime introduce controlli intermedi (session check, validazione payload, cleanup componenti in conflitto) e riduce side effect visivi o audio introdotti dagli asset importati.

Dopo il caricamento, il manager completa il setup comportamentale: animator idle, bind lip-sync e controller di sguardo/blink. Il comportamento embodied di questi componenti è discusso in 4.1.6; qui il punto progettuale è che il wiring avvenga subito dopo l'instanziazione, così ogni avatar entra in scena già coerente con il ciclo operativo previsto.

Infine, watchdog e cancellazione globale impediscono che un download lento lasci l'applicazione in lock permanente. Il meccanismo non elimina la causa del guasto, ma preserva la recoverability: l'utente può riprovare o tornare al menu senza riavviare la sessione.

² <https://docs.unity3d.com/Manual/class-ScriptableObject.html>

³ <https://docs.unity3d.com/Packages/com.unity.cloud.gltfast@6.10/manual/index.html>

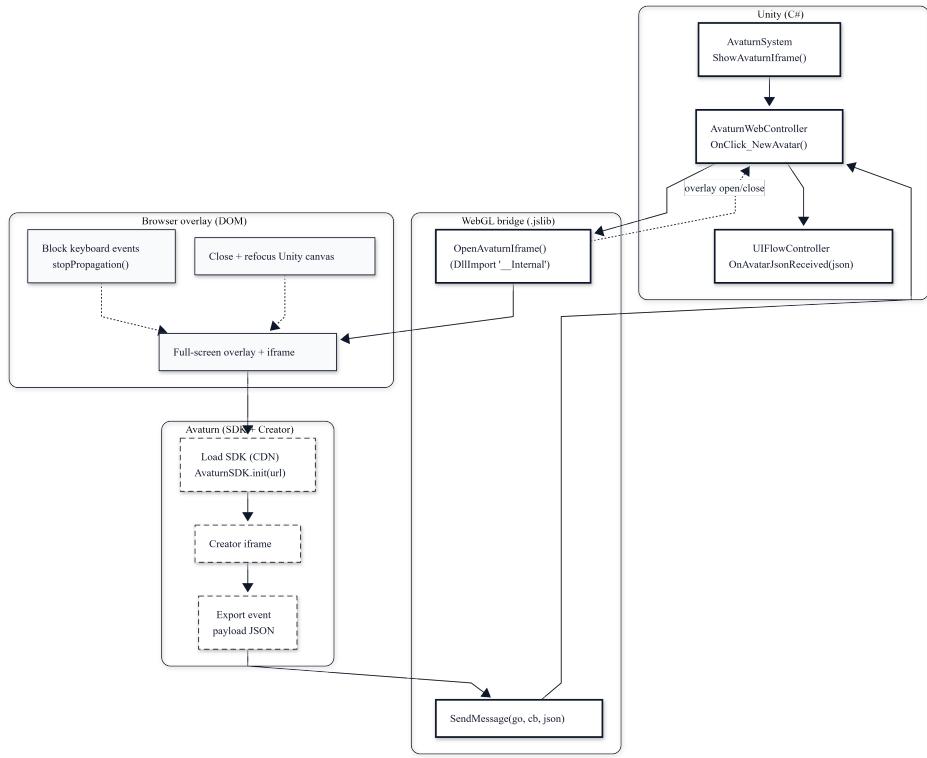


Fig. 4.3: Architettura dell'integrazione Avaturn in WebGL: AvaturnWebController invoca il bridge OpenAvaturnIframe in AvaturnBridge.jslib, che crea un overlay DOM e inizializza l'SDK Avaturn; l'evento di export ritorna a Unity tramite SendMessage con un payload JSON.

4.1.3 Integrazione Avaturn WebView/SDK nel client Unity

In WebGL l'editor Avaturn viene integrato come overlay DOM, non come componente Unity interno: il browser ospita l'iframe e Unity riceve solo il payload di export. Questa architettura è stata adottata perché in WebGL il confine naturale è Unity C# ↔ JavaScript, e forzare un modello "embedded" avrebbe aumentato complessità senza vantaggi operativi.

Il punto di ingresso lato Unity è AvaturnWebController. In editor apre l'URL esternamente, mentre in build WebGL invoca OpenAvaturnIframe via `[DllImport("__Internal")]`, secondo il pattern previsto da Unity per interoperare con script browser⁴ e coerente con la scena esempio WebGL dell'SDK ufficiale Avaturn [12], adattato per disaccoppiare il ciclo di vita dell'overlay dal-

⁴ <https://docs.unity3d.com/Manual/webgl-interactingwithbrowserscripting.html>

la FSM applicativa. Il contratto minimo (URL, game object target, callback method) resta sufficiente per disaccoppiare il ciclo UI dal bootstrap dell'SDK.

Il callback `OnAvatarJsonReceived` inoltra il JSON a `UIFlowController`, che resta il coordinatore in cui lo stato applicativo viene aggiornato. Questo passaggio evita duplicazione di logica tra web bridge e FSM: il bridge consegna eventi, la FSM decide transizioni e side effect.

In `AvaturnBridge.jslib` l'overlay viene creato, l'SDK viene inizializzato e l'evento `export` viene convertito in payload JSON inviato a Unity tramite `SendMessage`. Nel bridge restano anche le varianti di invio (istanze Unity esposte con nomi diversi) e la serializzazione di fallback, per rendere il comportamento robusto rispetto ai diversi template WebGL.

Gestire focus e input è parte integrante della soluzione: apertura overlay, blocco eventi verso Unity e refocus del canvas in chiusura prevengono stati in cui la tastiera resta agganciata al DOM o si perde tra i due contesti. Apertura/chiusura coordinata dell'overlay e refocus esplicito del canvas risolvono un problema UX concreto, non solo tecnico: evitare click di recupero dopo la chiusura dell'editor.

`AvaturnSystem` resta un wrapper di compatibilità: disabilita UI Avaturn non necessaria in WebGL e delega l'avvio al bridge JavaScript. Questo strato riduce l'accoppiamento con prefab terzi e mantiene la possibilità di sostituire il provider web senza riscrivere il flusso frontend.

Separare `AvaturnWebController`, bridge JavaScript e `UIFlowController` consente al browser di gestire l'editor in isolamento, lasciando a `AvatarManager` il consumo del payload di import come descritto in 4.1.2.

4.1.4 Acquisizione audio e input desktop/touch

Nel client WebGL l'acquisizione audio è trattata come sottosistema separato dalla UI. La separazione stabilizza la pipeline voce anche quando cambia la sorgente input (desktop o touch) e quando cambia il runtime (Unity nativo o browser).

L'architettura è a tre livelli: `AudioRecorder` come facade usata da `UIFlowController`, `WebGLAudioCapture` come bridge C#, e `AudioCapture.jslib` come implementazione browser. L'organizzazione a livelli riduce l'accoppiamen-

to: la UI invoca sempre le stesse operazioni di alto livello, mentre i dettagli di piattaforma restano confinati nel provider.

Nel percorso WebGL, l'interoperabilità usa `DllImport("__Internal")` e callback AOT-safe (AOT, Ahead-Of-Time: modalità di compilazione di Unity WebGL che impone vincoli sui tipi di delegate e callback utilizzabili a runtime)⁵. Il callback restituisce puntatore e lunghezza, poi i byte vengono copiati immediatamente in memoria gestita con `Marshal.Copy`. Il passaggio resta esplicito per evitare dipendenze dall'ownership del buffer Emscripten oltre la durata della chiamata.

La gestione permessi è centralizzata nel provider: richiesta anticipata, stato esposto al chiamante e prevenzione di richieste concorrenti. La centralizzazione riduce il fallimento del primo turno vocale dovuto a prompt browser tardivi e rende più chiaro, lato UI, se un errore dipende da autorizzazioni o dalla cattura.

Nel plugin JavaScript, `MediaRecorder` raccoglie il flusso audio e `decodeAudioData` lo converte in un `AudioBuffer`; il payload finale viene serializzato in WAV PCM standard e trasferito a Unity via heap Emscripten (`_malloc/_free`). La scelta di produrre direttamente un WAV compatibile con la pipeline STT/TTS evita conversioni aggiuntive nel backend.

Il sistema supporta sia modalità start/stop sia cattura a durata fissata. La scelta di mantenerle entrambe risponde a due esigenze diverse: turno conversazionale in tempo reale e acquisizione guidata durante setup. La semantica di quando invocarle resta nel livello UI, come discusso in 4.1.1.

La catena in Figura 4.4 mostra perché questa struttura è rilevante nel capitolo: validazione vocale (4.1.5) e `MainMode` condividono lo stesso contratto di output audio. Consolidare tale contratto in un solo punto riduce regressioni trasversali tra onboarding e conversazione.

4.1.5 Validazione del campione vocale

Lo stato `SetupVoice` completa l'onboarding dell'avatar verificando che il campione vocale registrato sia effettivamente una lettura del testo di riferimento mostrato a schermo, prima di salvare il file come voce di *voice cloning* per la sintesi successiva (Capitolo 3). Il controllo non ha finalità biometriche: serve a ridurre

⁵ <https://docs.unity3d.com/Manual/webgl-interactingwithbrowserscripting.html>

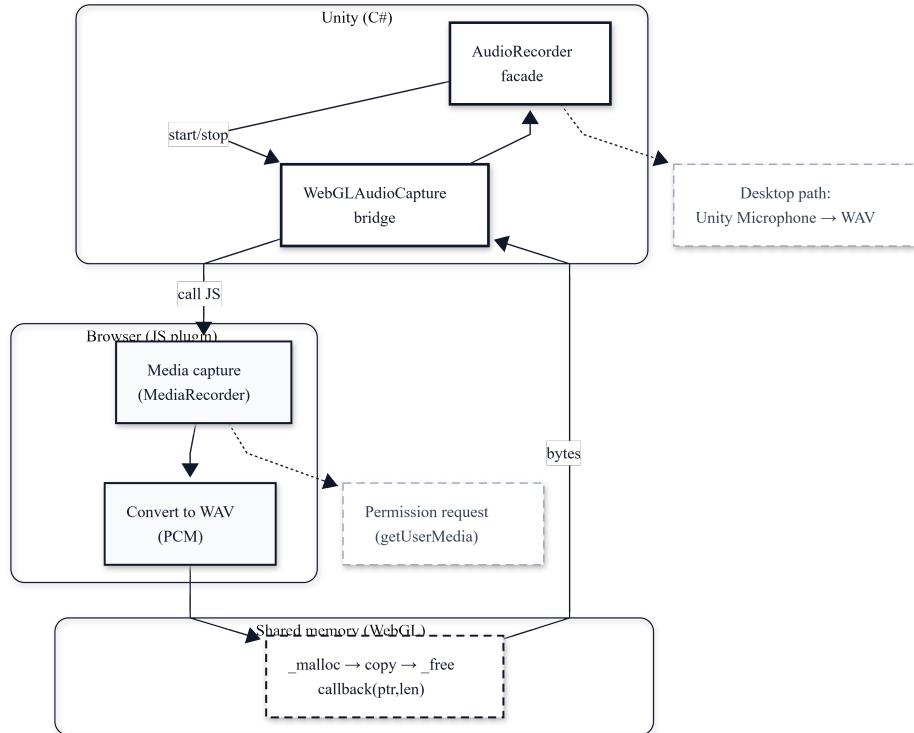


Fig. 4.4: Stack di acquisizione audio in WebGL: AudioRecorder delega al provider IAudioCaptureWebGL (WebGLAudioCapture), che richiama il plugin AudioCapture.jslib per cattura MediaRecorder e conversione WAV (header RIFF) con trasferimento dei byte via heap Emscripten.

casi pratici che degradano la qualità percepita, come registrazioni troppo brevi, audio vuoto, letture in una lingua diversa o campioni registrati mentre l’utente parla d’altro. La validazione basata su similarità testuale resta economica, deterministica e spiegabile in UI (percentuale di match), senza introdurre modelli addizionali o dataset specifici.

Il flusso parte dalla generazione della frase di riferimento `setupVoicePhrase`. `UIFlowController` costruisce la frase con una coroutine dedicata (`SetupVoicePhraseRoutine`) che impone una lunghezza minima di `setupVoiceTargetWords=32` parole e un margine massimo controllato da `setupVoiceWordSlack=10`. Se la generazione non produce un testo valido, il controller seleziona una frase di fallback predefinita, evitando che la UI resti bloccata su contenuti vuoti o su risposte anomale. Questo vincolo mantiene la stessa esperienza tra sessioni e rende il requisito della lettura più stabile:

all’utente viene sempre proposto un testo sufficientemente lungo da rappresentare consonanti, vocali e transizioni tipiche dell’italiano, ma non così esteso da rendere la fase di setup eccessivamente lenta.

L’acquisizione del campione avviene in modalità *push-to-talk*: l’utente tiene premuto il comando di registrazione (desktop: Space; touch: controllo hold dedicato) e al rilascio il controller avvia l’elaborazione asincrona. La routine `ProcessSetupVoiceRecording` richiama `audioRecorder.StopRecordingAsync` per ottenere i byte WAV e applica controlli minimi di consistenza (byte presenti e dimensione non nulla). In caso di fallimento, lo stato resta in `SetupVoice` e la UI comunica l’esito con messaggi esplicativi (ad esempio “Registrazione fallita. Riprova.”), evitando transizioni premature. Quando l’audio è disponibile, il client invia il file al micro-servizio di trascrizione tramite `PostWavToWhisper`, che effettua una POST su `/transcribe` includendo il campo `language="it"` e recupera la stringa `text` dal JSON di risposta. L’uso di Whisper come Speech-to-Text (STT) consente una trascrizione robusta in condizioni realistiche, rendendo praticabile la validazione anche con rumore e variazioni di dizione.[7]

Il cuore della validazione è il calcolo di una similarità tra la frase attesa e la trascrizione ottenuta. Il metodo `CalculateSimilarity(expected, actual)` implementa una combinazione pesata di due segnali complementari: la distanza di Levenshtein, che misura il numero minimo di inserzioni, cancellazioni e sostituzioni necessarie per trasformare una stringa nell’altra, e il coefficiente di Jaccard, che misura la sovrapposizione tra due insiemi di token. La prima cattura errori locali (inserzioni, cancellazioni e sostituzioni di caratteri) che tipicamente emergono da misallineamenti nella trascrizione; il secondo segnala se la trascrizione copre davvero il vocabolario della frase, riducendo casi in cui una parte consistente del testo viene omessa o riformulata.[13] In pratica, i due punteggi discriminano fallimenti diversi: la distanza di edit penalizza trascrizioni “quasi giuste ma sporche”, mentre Jaccard penalizza trascrizioni parziali o con lessico troppo distante.

Prima di confrontare i testi, il controller applica una normalizzazione che riduce variabilità non informativa. `NormalizeForCompare` converte a minuscolo, rimuove diacritici tramite normalizzazione Unicode (`FormD` e filtro dei caratteri `NonSpacingMark`) e sostituisce ogni simbolo non alfanumerico con spazio; una regex compatta poi spazi multipli e rifila i bordi. Questa normalizzazione evita

che punteggiatura, apostrofi o accenti compromettano eccessivamente lo score, soprattutto perché l'ASR può introdurre differenze minime (ad esempio “città” vs “citta”) che non indicano un errore sostanziale nella lettura. Dopo la normalizzazione, la frase viene rappresentata sia come stringa continua (per la similarità di sequenza) sia come insieme di token separati da spazi (per l'overlap lessicale).

La distanza di Levenshtein tra due stringhe normalizzate a e b viene calcolata con programmazione dinamica su una matrice $D \in \mathbb{N}^{(|a|+1) \times (|b|+1)}$, inizializzata sui bordi e aggiornata con la regola classica di minimo costo di edit:

$$D_{i,0} = i, \quad D_{0,j} = j$$

$$D_{i,j} = \min \left(D_{i-1,j} + 1, \ D_{i,j-1} + 1, \ D_{i-1,j-1} + \delta(a_i, b_j) \right)$$

dove $\delta(a_i, b_j) = 0$ se i caratteri coincidono e $\delta(a_i, b_j) = 1$ altrimenti. La distanza finale è $\text{lev}(a, b) = D_{|a|, |b|}$. Poiché una distanza assoluta cresce con la lunghezza, **UIFlowController** la trasforma in una similarità normalizzata in $[0, 1]$ dividendo per la lunghezza massima e invertendo il segno:

$$s_{\text{seq}}(a, b) = 1 - \frac{\text{lev}(a, b)}{\max(|a|, |b|)}.$$

In parallelo, il punteggio lessicale usa il coefficiente di Jaccard sui set di parole $W(a)$ e $W(b)$ ottenuti dallo split su spazi:

$$s_{\text{word}}(a, b) = \frac{|W(a) \cap W(b)|}{|W(a) \cup W(b)|}.$$

Il punteggio finale restituito da **CalculateSimilarity** è una media pesata clippata in $[0, 1]$:

$$s(a, b) = \text{clip}_{[0,1]} \left(0.6 \cdot s_{\text{seq}}(a, b) + 0.4 \cdot s_{\text{word}}(a, b) \right).$$

Questi coefficienti rappresentano un compromesso operativo: la parte di sequenza resta predominante perché la frase da leggere è unica e l'ordine dei caratteri contiene informazione utile; l'overlap lessicale conserva comunque un peso significativo per penalizzare omissioni e salti di intere porzioni, frequenti quando l'utente interrompe la lettura o quando il microfono cattura solo frammenti. Il

peso maggiore su s_{seq} riflette anche il comportamento tipico dell’ASR: Whisper tende a produrre errori locali di carattere (apostrofi, accenti, piccole sostituzioni) più che omissioni lessicali estese, che ricadono invece nel dominio di Jaccard.

La decisione di accettazione confronta lo score con la soglia `setupVoiceMinSimilarity=0.7f`. Il controller traduce lo score in percentuale (`Match %`) e aggiorna `setupVoiceStatusText` per rendere chiaro l’esito senza dover interpretare valori numerici. La soglia al 70% è una scelta empirica di progetto (configurabile da Inspector), utile a bilanciare tolleranza agli errori ASR e controllo qualità del campione, ma non deriva da benchmark o da una taratura statistica su dataset esterni. Se $s < 0.7$, il flusso termina nello stesso stato e richiede una nuova registrazione; questo controllo evita di salvare un riferimento vocale potenzialmente incoerente, che avrebbe effetti a catena sulla qualità del TTS e sulla percezione d’identità dell’avatar. Quando invece $s \geq 0.7$, `ProcessSetupVoiceRecording` procede con il salvataggio del campione vocale inviando il WAV al servizio TTS tramite `PostWavToCoqui` (endpoint `/set_avatar_voice`), associandolo all’`avatar_id` corrente, e avvia la generazione delle frasi di attesa (`/generate_wait_phrases`). In questo modo la validazione funge da guard-rail che protegge il profilo vocale usato da Coqui XTTS v2 per la sintesi zero-shot, riducendo la probabilità di profili di bassa qualità dovuti a input non controllato.[9]

Un aspetto rilevante è la gestione degli errori e delle cancellazioni durante la pipeline asincrona. Il controller mantiene uno stato di annullamento (`setupVoiceCancelling`) e, dopo ogni step (stop recording, trascrizione, match, upload TTS), verifica se l’utente abbia richiesto di interrompere l’operazione; in tal caso esce dalla routine senza applicare side effect ulteriori. In parallelo, gli errori dei servizi (Whisper o Coqui) vengono riportati con messaggi distinti e accompagnati da feedback sonoro (`PlayErrorClip`), migliorando la diagnosi durante test e limitando i casi in cui l’utente non capisce se il problema sia di rete, permessi microfono o contenuto della registrazione. In un onboarding vocale questo dettaglio è decisivo: l’utente accetta più facilmente di ripetere una registrazione quando il sistema esplicita perché la richiesta è fallita e quando il criterio di successo è osservabile.

4.1.6 Comportamento embodied: lip sync e sguardo idle

Nel client SOULFRAME il comportamento embodied non è un effetto accessorio dell’animazione idle, ma un sottosistema con wiring dedicato subito dopo l’instanziazione dell’avatar. In `AvatarManager`, il setup viene avviato con `SetupLipSyncNextFrame` e `SetupIdleLookNextFrame`, che chiamano `lipSyncBinder.Setup(avatarRoot)` e `idleLook.Setup(avatarRoot)` al frame successivo. La separazione tra i livelli è necessaria perché lip sync e sguardo dipendono da segnali runtime (audio TTS, input utente, stato conversazionale) che non possono essere modellati in modo robusto con una clip preauthorata.

Il nodo centrale del lip sync è `AvaturnULipSyncBinder`. In fase di setup, il binder esegue una scansione runtime dei `SkinnedMeshRenderer`, individua i blendshape compatibili con i visemi e costruisce una mappa viseme → indice blendshape usata durante l’aggiornamento. L’applicazione dei pesi avviene in `LateUpdate`, pilotata dagli eventi `OnLipSyncUpdate(LipSyncInfo)` di `uLipSync`⁶: in questo modo la bocca segue il segnale fonetico corrente invece di una timeline fissa. Il binding runtime è preferibile a una tabella hardcoded perché gli avatar Avaturn non garantiscono naming e disponibilità dei blendshape identici tra profili diversi; un mapping statico è fragile e aumenta i casi di avatar “parlante” ma visivamente fermo.

Il lip sync è inoltre agganciato direttamente al percorso audio TTS usato in conversazione: `UIFlowController.BeginMainMode` recupera la sorgente con `GetOrCreateLipSync AudioSource`, privilegiando il componente `uLipSync AudioSource`; i chunk PCM vengono poi riprodotti su quella sorgente tramite `PcmChunkPlayer`. Questo aggancio diretto mantiene allineate pipeline audio e pipeline visiva, evitando disaccoppiamenti tra voce sintetizzata e animazione labiale.

`ULipSyncProfileRouter` completa la catena scegliendo il profilo `uLipSync` coerente con il genere del profilo avatar, senza duplicare logica nella UI. Nel flusso corrente `ApplyGender(...)` viene invocato in `AvatarManager` sia all’import di un nuovo avatar sia al caricamento di un avatar salvato.

Per sguardo e micro-movimenti, `AvaturnIdleLookAndBlink` gestisce blink, head tilt e target di look di base, mentre in `MainMode`

⁶ <https://github.com/hecomi/uLipSync>

`UpdateMainModeMouseLook()` aggiorna `SetExternalLookTarget(...)` come override esterno quando l’utente interagisce con il puntatore (mouse, e su touch il tocco primario). Lo stesso componente riceve `OnLipSyncUpdate(LipSyncInfo)` per distinguere parlato, ascolto e idle in `LateUpdate`. Questo sistema resta separato dall’animatore idle perché i due livelli risolvono problemi diversi: l’animatore governa la posa generale, il look controller governa segnali attentivi reattivi e a bassa latenza.

Questi segnali hanno un effetto concreto sulla percezione dell’interazione. Piccoli movimenti di sguardo, ascolto e blink evitano l’effetto di staticità quando non c’è parlato e aiutano a leggere l’avatar come presenza attiva, non come modello fermo. Nei turni conversazionali più lenti, questa continuità visiva riduce la sensazione di “vuoto” tra un input e la risposta.

4.1.7 Post-processing, rings di stato e feedback di selezione

Nel client, post-processing, rings e feedback di selezione sono trattati come un sottosistema unico, non come effetti isolati, così da mantenere leggibilità dello stato e coerenza percettiva tra avatar, sfondo e controlli UI. Il prefisso `PS2*` nei nomi dei componenti è una convenzione interna di naming; nel testo si usano etichette descrittive (post-processing, rings, feedback di selezione). Il riferimento estetico resta quello dei menu PlayStation 2, ma la lettura operativa passa da segnali visivi concreti.

Il blocco di post-processing inizializza un `Volume` globale URP e governa `Bloom` con override runtime. Il bootstrap esplicito è preferibile a profili statici distribuiti sulle camere perché il progetto deve restare stabile tra editor, build desktop e WebGL, con un coordinatore centrale per applicare parametri coerenti. La pipeline attiva il post-processing sulle camere tramite `UniversalAdditionalCameraData.renderPostProcessing`; in WebGL viene applicato un profilo alleggerito per contenere il costo GPU senza perdere l’identità visiva complessiva.

I rings sono gestiti da `PS2BackgroundRings`, controllato da `UIFlowController` tramite `SetOrbitSpeedMultiplier(...)` e da transizioni animate della trasformazione. La paletta resta costante per mantenere continuità tra identità visiva e segnali di stato. La mappatura operativa è esplicita: durante l’attesa di inizializ-

zazione di Coqui-TTS in `Boot` i rings rallentano e il post-processing abilita il pulse di scatter; nel *download state* (e nelle operazioni bloccanti di setup voce/memoria) la velocità aumenta per comunicare attività in corso; nello stato normale si torna al moltiplicatore base. Questa mappatura rende leggibile la pipeline senza introdurre testo aggiuntivo in UI.

La posizione è anch'essa guidata dallo stato: in `MainMenu/Boot` i rings usano la posizione di default, durante le operazioni in `SetupVoice` e `SetupMemory` si ancorano a target dedicati (con varianti touch), mentre negli stati conversazionali vengono spostati fuori campo con un offset relativo alla camera. Lo spostamento usa un'interpolazione morbida; durante il moto, il controller dei rings rileva la velocità del centro e passa temporaneamente a un comportamento "a scia", in cui le particelle si dispongono in fila dietro la direzione di movimento. Sul piano della resa, le particelle sono billboard (orientate alla camera) e, quando il sistema non è in `RectTransform`, anche il transform radice viene riallineato alla camera per mantenere coerenza visiva.

Il feedback di selezione dei controlli è implementato con `SelectableBlink`: il componente applica una modulazione luminosa al controllo attivo e, sui testi TMP, abilita anche un glow coerente con la paletta generale. Questo feedback sostituisce highlight più neutri ma meno informativi, confermando in modo immediato quale elemento sta ricevendo input da tastiera o gamepad/touch. L'effetto combinato dei blocchi produce un sistema di feedback unico: comunica stato, conferma azione e mantiene continuità stilistica tra UI e avatar.

4.1.8 MainMode conversazionale

`MainMode` realizza il ciclo conversazionale completo con una scelta architettonica precisa: separare lo stato UI (ascolto, processing, speaking) dal dettaglio delle chiamate rete. Questa distinzione mantiene il turn-taking prevedibile anche quando STT, RAG e TTS hanno latenze variabili.

All'ingresso, `BeginMainMode()` riallinea il contesto e verifica prerequisiti minimi (voce e memoria avatar). La verifica iniziale evita di avviare conversazioni "stateless" e rende esplicito il legame con le fasi di onboarding descritte in 4.1.5 e 4.1.2. In parallelo viene attivato il comportamento non verbale dell'avatar, mentre il controllo microfono riusa lo stack descritto in 4.1.4.

Il turno vocale resta PTT sia su desktop sia su touch: cambia il binding input, non la semantica applicativa. La convergenza sulle stesse entrypoint (start/stop) evita biforazioni nel flusso e garantisce che errori, messaggi e fallback siano identici tra piattaforme.

Il flusso conversazionale è articolato in tre fasi. La trascrizione invia WAV a Whisper con `UnityWebRequest.Post`⁷ e aggiorna UI solo dopo deserializzazione valida; questo vincolo privilegia robustezza e rende esplicito il punto di fallimento in caso di rete o payload non conforme. L'adozione di Whisper resta coerente con l'obiettivo di tolleranza a rumore e variazioni di dizione[7].

In fase RAG si costruisce il payload conversazionale, si invia `/chat` e si mostra una risposta sanitizzata prima della sintesi. Mantenere questa fase separata dalla TTS è utile perché il testo risposta è un output autonomo dell'interazione e deve restare leggibile anche in caso di problemi audio. La chat non genera a vuoto: prima recupera frammenti dalla memoria dell'avatar, poi costruisce il prompt su quella base secondo il paradigma *Retrieval-Augmented Generation*[8].

Per la sintesi si adotta streaming incrementale: i chunk arrivano tramite `PcmStreamDownloadHandler` (`DownloadHandlerScript`)⁸ e vengono riprodotti da `PcmChunkPlayer`. Questo schema riduce la latenza percepita senza attendere l'audio completo; il fallback audio di cortesia che copre la finestra iniziale è descritto in sottosezione 4.2.3.

Strettamente collegata è la gestione dell'interruzione: `InterruptMainModeSpeech()` abortisce richieste attive, ferma playback e invalida sessioni correnti. Questo comportamento rende il turn-taking effettivamente interattivo, perché i chunk tardivi vengono scartati e non possono "riaccendersi" su uno stato già superato.

La pipeline in Figura 4.5 sintetizza la decisione principale: trattare STT, RAG e TTS come stadi indipendenti ma coordinati da uno stato unico, così da bilanciare reattività UI e continuità vocale. La sintesi in streaming si integra inoltre con modelli di voice cloning come XTTS adottato nel backend[9].

`MainMode` usa lo stesso wiring avatar descritto in 4.1.6: controllo sguardo, gestione idle e regole touch contestuali vengono applicati senza duplicare logica. Questa continuità evita un effetto visivo evidente in ingresso conversazionale:

⁷ <https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.html>

⁸ <https://docs.unity3d.com/ScriptReference/Networking.DownloadHandlerScript.html>

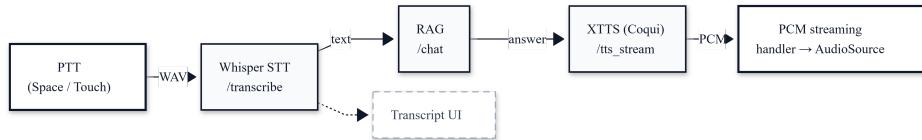


Fig. 4.5: Pipeline conversazionale in MainMode: PTT → WAV → Whisper/transcribe → RAG/chat → Coqui/tts_stream → PcmStreamDownloadHandler → PcmChunkPlayer → AudioSource.

avatar correttamente caricato ma ancora immobile, con sguardo non agganciato, nei primi frame prima dell'attivazione dei controller comportamentali.

4.2 Implementazione backend

Il backend di SOULFRAME è organizzato come architettura a micro-servizi in Python basata su FastAPI, con quattro servizi indipendenti esposti su porte dedicate: *Speech-to-Text* (STT) su 8001, *Retrieval-Augmented Generation* (RAG) su 8002, servizio di *asset avatar* su 8003 e *Text-to-Speech* (TTS) su 8004; a supporto opera un'istanza locale di Ollama sulla porta 11434 per le funzioni di embedding e chat LLM. L'avvio dei processi è automatizzato tramite lo script `ai_services.cmd`, che invoca `uvicorn` per ciascun servizio e fornisce comandi operativi di `start/stop/restart`; prima di avviare nuovi processi, lo script verifica inoltre la presenza di porte già in ascolto per evitare duplicazioni e conflitti. Le dipendenze sono centralizzate in un unico `requirements.txt`, con versioni fissate per mantenere riproducibile l'ambiente tra sviluppo e deploy. L'adozione di FastAPI è motivata dal supporto nativo all'*async I/O*, dalla validazione dei payload tramite Pydantic⁹ e dalla generazione automatica della specifica OpenAPI e della relativa UI interattiva¹⁰. La scomposizione in servizi a responsabilità mirata segue i principi di *bounded context* e di deploy indipendente tipici dei micro-servizi, favorendo isolamento dei failure e aggiornabilità selettiva dei componenti [14].

La Figura 4.6 illustra l'architettura complessiva del backend con le porte dei servizi e le dipendenze principali.

⁹ <https://docs.pydantic.dev/>

¹⁰ <https://fastapi.tiangolo.com/>

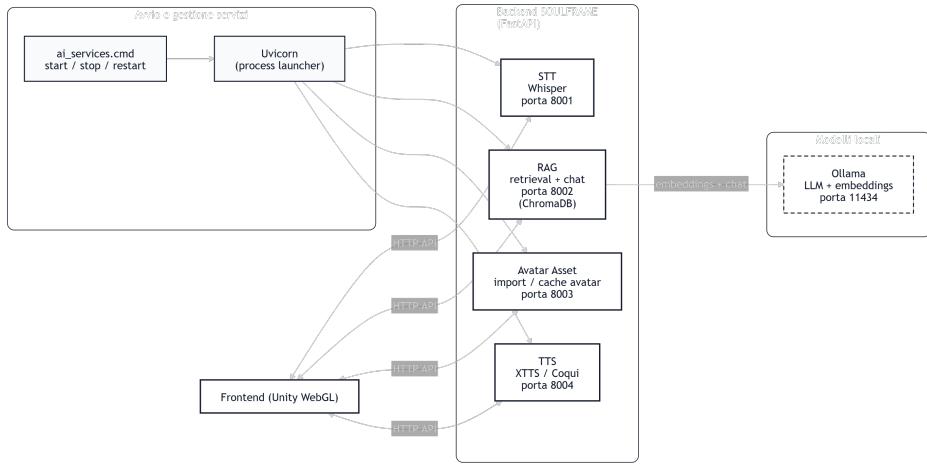


Fig. 4.6: Architettura dei micro-servizi backend: i quattro servizi FastAPI (Whisper, RAG, Avatar Asset, TTS) comunicano su porte dedicate; il servizio RAG si appoggia a Ollama (porta 11434) per embedding e chat LLM.

4.2.1 Servizio STT (Whisper)

Il servizio di *Speech-to-Text* (STT) è implementato nel file `whisper_server.py` come applicazione FastAPI, avviata tramite Uvicorn sulla porta 8001 (gestita dallo script operativo `ai_services.cmd`). All'avvio del processo si carica in memoria un modello Whisper mediante `whisper.load_model(MODEL_NAME)`, dove `MODEL_NAME` è ricavato dalla variabile d'ambiente `WHISPER_MODEL` (default "small"). Il parametro consente di selezionare profili noti (`tiny`, `base`, `small`, `medium`, `large`) in base al compromesso tra accuratezza e latenza: modelli più piccoli riducono i tempi di trascrizione e il consumo di memoria, mentre modelli più grandi tendono a migliorare la qualità in presenza di rumore e parlato complesso. Nel setup usato per lo sviluppo locale Windows, con circa 12 GB di VRAM e vincolo forte sui tempi di risposta turn-by-turn, il profilo `small` è risultato il compromesso più stabile. Nel setup server Ubuntu testato, con circa 24 GB di VRAM, è stato invece usato `medium`. Nel progetto la dipendenza è fissata in `requirements.txt`, così da rendere riproducibile l'ambiente di esecuzione.

L'endpoint principale è `POST /transcribe`. Esso riceve un campo `file` (campo `UploadFile`) e un campo `Form language`, con default "it". Il servizio non impone una whitelist esplicita dei formati, ma si appoggia alla catena di decoding supportata da Whisper, che in pratica consente l'uso di formati comuni quali WAV,

MP3, M4A, FLAC e OGG.¹¹ Prima della chiamata a `model.transcribe(...)` il parametro `language` viene normalizzato per gestire in modo robusto anche payload `Form` multivalore; la gestione del file temporaneo è racchiusa in un flusso con `cleanup` in `finally`, così da evitare accumuli su disco in caso di errore.

Per supportare l'integrazione con il client WebGL, l'applicazione abilita un middleware CORS con `allow_origins=["*"]`, necessario quando il frontend è servito da un dominio o da una porta diversa rispetto a `8001`. Il servizio espone inoltre GET `/health`, che restituisce `{"ok": True, "model": MODEL_NAME}` e viene utilizzato come health-check rapido in fase di bootstrap.

Le proprietà architetturali di Whisper e la motivazione della sua adozione sono discusse in §2.2; qui conta che la robustezza *zero-shot* rende la trascrizione praticabile con microfoni eterogenei senza fine-tuning.

4.2.2 Servizio RAG e memoria per avatar

Il servizio RAG e memoria è implementato in `rag_server.py` come applicazione FastAPI in ascolto sulla porta `8002`. Il componente realizza un paradigma di *Retrieval-Augmented Generation* (RAG), in cui la generazione testuale si appoggia a una memoria non-parametrica persistente per ridurre incoerenze tra turni e mantenere informazioni specifiche dell'avatar [8]. Ogni avatar deve poter mantenere ricordi distinti (note, preferenze, stile) senza contaminare gli altri profili; per questo la persistenza è isolata per `avatar_id`: viene creato un database ChromaDB dedicato in `rag_store/<avatar_id>/`, istanziato tramite `chromadb.PersistentClient(path=...)` e riusato in cache con un lock globale per evitare riaperture concorrenti.¹² All'interno del database dell'avatar si mantiene una singola collezione `memory`, ottenuta con `get_or_create_collection(name="memory")`, che memorizza documenti testuali, metadati e vettori embedding. La separazione per cartella rende strutturale l'isolamento: azzerare o migrare un profilo non impatta sugli altri e il concetto di “memoria” resta allineato all'identità dell'avatar selezionato nel frontend.

Gli embedding non sono calcolati nel processo FastAPI, ma delegati a un runtime locale Ollama.¹³ Il servizio invoca `/api/embed` sul

¹¹ <https://github.com/openai/whisper>

¹² <https://docs.trychroma.com/>

¹³ <https://ollama.com/docs>

loopback (`OLLAMA_HOST`, default `http://127.0.0.1:11434`) usando il modello `nomic-embed-text` (`EMBED_MODEL`), supportando sia richieste singole sia batch. La generazione conversazionale usa invece `/api/chat` con `llama3:8b-instruct-q4_K_M` (`CHAT_MODEL`) e opzioni configurabili da variabili d'ambiente (ad esempio `temperature=0.45`).

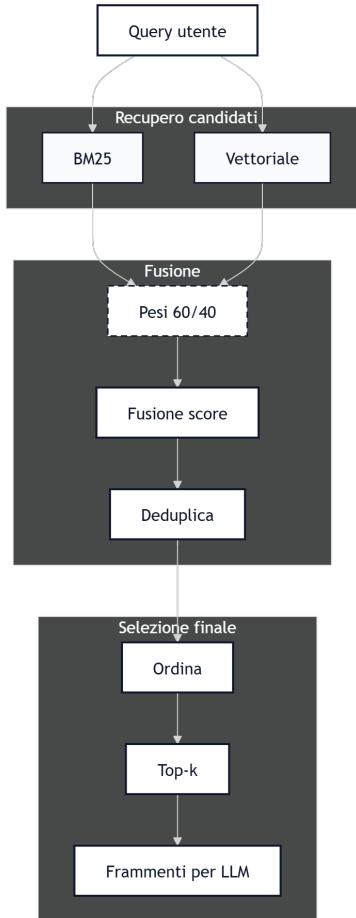


Fig. 4.7: Ricerca ibrida nel servizio RAG: pesatura BM25/vettoriale (60/40), fusione e deduplicazione dei frammenti per il prompt LLM.

Nel setup locale Windows la scelta del profilo quantizzato a 4-bit è legata al vincolo di circa 12 GB di VRAM, condiviso con Whisper e XTTS: riduce il costo memoria mantenendo una qualità adeguata per turni conversazionali brevi. Nel setup Ubuntu testato, con circa 24 GB di VRAM, la configurazione resta più permissiva. La distinzione è resa esplicita anche a livello operativo: in sviluppo Windows, `ai_services.cmd` imposta `CHAT_MODEL=llama3:8b-instruct-q4_K_M`; nel deploy Ubuntu, `setup_soulframe_ubuntu.sh` inizializza `CHAT_MODEL=llama3.1:8b` nel file `soulframe.env`. Le due varianti restano configurabili senza modificare il client, perché il contratto HTTP tra Unity e servizio RAG non cambia. La delega a Ollama mantiene il micro-servizio focalizzato sull'orchestrazione (retrieval, composizione del prompt, gestione persistenza) e rende sostituibili i modelli lato Ollama senza modificare l'API esposta al client.

La fase di retrieval adotta una ricerca ibrida che combina segnali lessicali e semantici. In `_hybrid_search` si esegue prima una query vettoriale su ChromaDB per recuperare un insieme di candidati (`top_k*3`, con limite massimo 100); su questi candidati viene poi applicato un reranking BM25 (Best Match 25, un modello di ranking lessicale che pondera frequenza dei termini e lun-

ghezza del documento) [15]. Gli score BM25 e quelli vettoriali (derivati da $1 - \text{distance}$) vengono normalizzati e combinati con peso 0.6 per BM25 e 0.4 per la componente vettoriale, producendo una graduatoria finale da cui si estraggono i `top_k` frammenti. Prima dell'inserimento nel contesto del prompt, i chunk sovrapposti vengono deduplicati con una similarità di sequenza, riducendo ripetizioni e ridondanza.

Gli endpoint principali riflettono questo ruolo di “memoria per-avatar” e di orchestrazione conversazionale. `POST /remember` salva un testo nella memoria dell'avatar: il contenuto viene normalizzato (`clean_text`), validato con filtri anti-garbage e indicizzato calcolando l'embedding via Ollama; ogni documento viene corredata di metadati scalari (sorgente, timestamp, `avatar_id`). `POST /recall` fornisce una via di diagnostica: calcola l'embedding della query e restituisce documenti e metadati recuperati tramite ricerca ibrida, con fallback automatico alla sola similarità coseno su ChromaDB (query vettoriale standard) se il modulo BM25 non è disponibile o restituisce errore. Il percorso conversazionale è invece `POST /chat`, che applica retrieval, costruisce il prompt e invoca `/api/chat` su Ollama, restituendo `{"text": answer, "rag_used": ..., "auto_remembered": ...}` al client. A supporto del flusso UI, `GET /avatar_stats` espone conteggio documenti e stato memoria per `avatar_id`, così da verificare rapidamente i prerequisiti conversazionali.

La composizione del prompt in `/chat` implementa un meccanismo esplicito anti-allucinazione. Il server costruisce un `system prompt` che impone di parlare come avatar in prima persona e vieta forme meta (ad esempio riferimenti al “contesto” o all’“utente”), oltre a rimuovere indicazioni sceniche e prefissi di parlante. Quando la flag `RAG_ENFORCE_GROUNDED` è attiva, il prompt aggiunge regole di grounding: non inventare fatti non presenti in memoria e dichiarare in modo trasparente eventuali conflitti tra ricordi. Il `user prompt` contiene invece tre blocchi: (i) `MEMORIA RAG` con frammenti e sorgenti sintetizzate in `_build_context_from_docs`, (ii) un blocco dedicato ai tratti di stile, (iii) il testo dell'utente. Dopo la generazione, l'output viene ripulito da

`_sanitize_chat_answer`, che rimuove etichette di parlante e stage directions (parentesi, asterischi) e normalizza spazi e punteggiatura, mantenendo il testo più compatibile con la resa vocale e con i vincoli del frontend.

Accanto al vincolo di grounding “usa solo i frammenti”, il servizio introduce una strategia di memorizzazione automatica (*auto-remember*) per ridurre la perdita di informazioni esplicitamente richieste dall’utente. In `/chat` viene rilevato l’intento esplicito di memorizzazione (ad esempio “ricorda che...” o “remember that...”) e, quando presente, il contenuto viene indicizzato come nuova memoria dell’avatar. Se l’auto-remember è avvenuto, il prompt utente include una nota di sistema che richiede una breve conferma esplicita del fatto che l’informazione verrà ricordata. Questa euristica non sostituisce il retrieval, ma riduce la probabilità che preferenze o vincoli enunciati in modo diretto vengano persi tra turni.

Il servizio distingue inoltre tra memoria fattuale e memoria di stile, introducendo una forma semplice di *persona extraction*. In fase di chat, oltre al retrieval standard per la query, il server esegue un recupero aggiuntivo mirato allo stile e trasforma i frammenti selezionati in “cues” sintetici da fornire al modello come vincolo morbido: l’obiettivo è replicare tratti linguistici quando presenti, senza degradare l’accuratezza dei contenuti.

L’ingestione della memoria avviene tramite POST `/ingest_file`, che supporta PDF, immagini e testo. Per i PDF si adotta OCR sistematico: ogni pagina viene renderizzata con PyMuPDF (`fitz`) e convertita in immagine prima di applicare `pytesseract` con lingua configurabile (`RAG_OCR_LANG`, default `ita+eng`); il testo OCR viene poi linearizzato per ridurre frammentazione e migliorare la similarità semantica. Le immagini (`.png/.jpg/.webp` e varianti) vengono processate con OCR diretto; i file testuali vengono decodificati con fallback di encoding e normalizzati. Il testo estratto viene segmentato con `chunk_text` usando parametri di lunghezza (`RAG_CHUNK_CHARS`) e overlap; i chunk troppo piccoli o “garbage” vengono scartati e i duplicati intra-file rimossi. Gli embedding vengono calcolati in batch da 16 (per bilanciare throughput e memoria) e poi inseriti in ChromaDB in batch controllati, limitando sia tempi di rete verso Ollama sia overhead di scrittura.

Per completare l’ingest multimediale, POST `/describe_image` consente una descrizione semantica di immagini tramite Gemini Vision quando configurato (chia-

ve e SDK disponibili); la descrizione può essere opzionalmente salvata in memoria con `source_type="image_description"`, permettendo di trasformare contenuto visivo non facilmente OCRizzabile in testo recuperabile. La pulizia operativa è infine gestita da `POST /clear_avatar`, che supporta una cancellazione “soft” (eliminazione della collezione `memory`) e una cancellazione “hard” (rimozione della directory persistente `rag_store/<avatar_id>`), gestendo in modo robusto gli errori di file system durante il reset.

4.2.3 Servizio TTS e streaming audio

Il servizio di *Text-to-Speech* (TTS) è implementato in `coqui_tts_server.py` come applicazione FastAPI avviata su porta `8004`. Il server carica un modello XTTS v2 tramite la libreria Coqui TTS¹⁴, con nome modello configurabile da variabile d’ambiente `COQUI_TTS_MODEL` (default `tts_models/multilingual/multi-dataset/xtts_v2`). L’obiettivo applicativo è offrire una sintesi multilingua con *voice cloning* in modalità zero-shot, cioè condizionata da un breve riferimento vocale associato all’avatar. In fase di inizializzazione, il caricamento viene eseguito da `_ensure_loaded()`, che seleziona automaticamente il device: se `torch.cuda.is_available()` è vero, viene preferita la GPU (`cuda`), altrimenti si ricade su `cpu`. È prevista anche una forzatura esplicita tramite `COQUI_TTS_DEVICE`. Nel caso in cui il modello sia stato inizializzato su GPU ma durante la sintesi emergano errori riconducibili a CUDA (ad esempio *out of memory*), il server tenta un fallback automatico su CPU, riducendo la probabilità di fault irreversibili a scapito della latenza.

Prima della sintesi, il testo viene normalizzato con la funzione `_clean_tts_text`. Il preprocessing rimuove simboli che potrebbero essere vocalizzati in modo indesiderato (ad esempio caratteri decorativi e parentesi), elimina virgolette e sequenze di punteggiatura ridondante, converte ellissi e trattini lunghi in pause più stabili, e normalizza spazi multipli. L’intento è stabilizzare la prosodia e prevenire letture “letterali” di markup o segni grafici, mantenendo tuttavia la punteggiatura utile come indizio per pause e intonazione. Il modello viene invocato con parametri di controllo della generazione configurabili da varia-

¹⁴ <https://docs.coqui.ai/>

bili d'ambiente (ad esempio temperatura e penalità di ripetizione), così da tarare stabilità vocale e naturalezza senza modificare la logica applicativa.

L'endpoint `POST /tts` fornisce la sintesi completa in un'unica risposta. Il client invia `text` e, opzionalmente, `avatar_id`, `language` e un file `speaker_wav`. Il server risolve il riferimento vocale secondo una priorità esplicita: se è presente un `speaker_wav` in upload, questo viene preprocessato e utilizzato; in caso contrario si usa `backend/voices/avatars/<avatar_id>/reference.wav` se disponibile, altrimenti un riferimento di default. Se richiesto (`save_voice=true`), l'upload viene anche salvato come riferimento persistente dell'avatar.

La sintesi genera un array di campioni in virgola mobile, che viene validato (non nullo e non troppo corto) e serializzato in `audio/wav` PCM16 (PCM, Pulse-Code Modulation a 16 bit: formato grezzo dell'audio digitale non compresso) tramite `soundfile`, restituendo un `Response` con header informativi su identità avatar e profilo vocale usato.

L'endpoint `POST /tts_stream` introduce invece una modalità di streaming pensata per ridurre la latenza percepita dal client. Il server segmenta il testo in chunk (`_split_text`) quando `split_sentences=true`, con limite `max_chunk_chars` e split preferenziale su punteggiatura. La risposta è una `StreamingResponse` con `media_type="audio/wav"`: lo stream emette prima un header WAV di 44 byte costruito da `_wav_header`, seguito da blocchi PCM16 raw prodotti chunk-by-chunk (`_iter_pcm_stream`). In questo protocollo l'header dichiara sample rate e formato, mentre il campo `data_size` è posto a zero, rendendo possibile iniziare

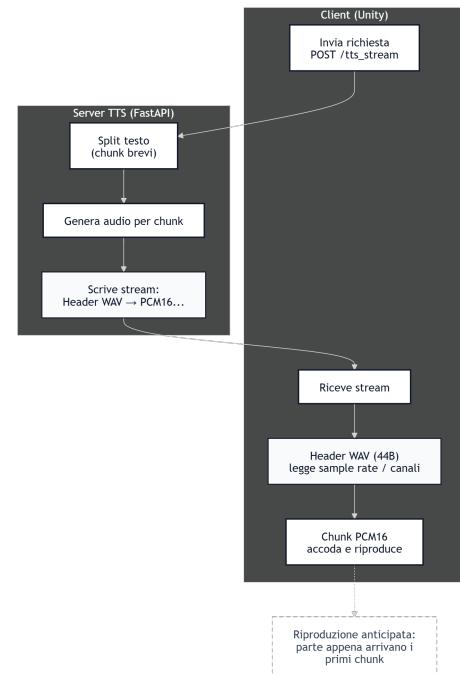


Fig. 4.8: Protocollo di streaming audio TTS: header WAV iniziale, chunk PCM in tempo reale e riproduzione anticipata prima del completamento della sintesi.

la riproduzione senza conoscere la durata totale. Il client può quindi avviare la riproduzione appena ricevuti i primi chunk, senza attendere il completamento della sintesi complessiva. Eventuali errori su singoli chunk non interrompono l'intero flusso: il server registra un warning e prosegue, e in caso di mancata emissione di audio restituisce un breve silenzio per mantenere lo stream valido.

La gestione del riferimento vocale per-avatar è esposta esplicitamente da `POST /set_avatar_voice`. L'endpoint riceve `avatar_id` e un `speaker_wav` e lo salva come `reference.wav` nella directory dedicata dell'avatar, dopo una fase di validazione e preprocess: lettura con `soundfile`, conversione a mono, rimozione del silenzio ai bordi quando abilitata, limitazione della durata massima e normalizzazione del picco. Questa pipeline rende il riferimento più stabile e contiene i casi in cui un file problematico (troppo corto, rumoroso o con silenzio prolungato) degrada la qualità di condizionamento.

Nel ciclo di vita dell'avatar, il client usa anche `GET /avatar_voice` per verificare presenza e dimensione del riferimento e `DELETE /avatar_voice` per rimuoverlo durante le operazioni di reset.

Per gestire la latenza iniziale tipica della sintesi neurale, il servizio include una strategia di pre-caching delle *wait phrases*. `POST /generate_wait_phrases` sintetizza un insieme finito di brevi frasi (ad esempio “Hm”, “Aspetta”, “Un secondo”) e le salva come `wait_<key>.wav` nella cartella dell'avatar. L'endpoint `GET /wait_phrase` recupera una frase già presente; se il file non è disponibile ma esiste un riferimento vocale valido, la frase viene sintetizzata on-demand e memorizzata, con un meccanismo di compatibilità verso un path legacy. Questo caching consente al frontend di riprodurre un feedback immediato mentre lo streaming TTS prepara i primi chunk.

Infine, per ridurre la penalità del primo utilizzo, il server può eseguire warmup e preload all'avvio¹⁵: viene sintetizzata una frase breve usando un riferimento vocale risolto per l'avatar, così da inizializzare i componenti di inferenza e, quando disponibile, la GPU prima del primo turno utente. Warmup e preload riducono la latenza percepita nel primo scambio. La scelta di XTTS v2 è coerente con un'impostazione zero-shot multilingua: il modello introduce un conditioning encoder

¹⁵ Il warmup all'avvio è configurabile tramite variabili d'ambiente (ad esempio `COQUI_WARMUP_ON_STARTUP`).

e meccanismi di quantizzazione e vincoli di consistenza del parlante, ottenendo risultati competitivi e supporto su 16 lingue [9].

4.2.4 Servizio asset avatar

Il servizio *asset avatar* è implementato in `avatar_asset_server.py` come applicazione FastAPI in ascolto sulla porta `8003`. Il componente ha responsabilità strettamente ingegneristiche: importare modelli 3D in formato glTF Binary (`.g1b`), mantenerne un catalogo persistente con metadati essenziali e servirli al client tramite endpoint HTTP. Il servizio è volutamente privo di dipendenze GPU e di logiche di inferenza: l'elaborazione è I/O-bound (download e file system) e il footprint runtime rimane minimale, rendendo l'istanza adatta anche a deployment economici o ambienti di test.

L'endpoint `GET /avatars/list` espone l'inventario degli avatar disponibili. La lista include sia modelli locali predefiniti (raggruppati in `LOCAL_MODELS`), sia i modelli importati dinamicamente, unificati in un unico payload JSON con metadati utili al frontend (identificatore, nome visualizzato, path relativo o URL di serving e sorgente). Il client può quindi popolare la libreria avatar senza dipendere dalla persistenza locale del browser, delegando al backend la “fonte di verità” della catalogazione. La logica di listaggio filtra i record corrotti e normalizza i campi per mantenere l'API stabile anche quando la struttura interna del catalogo evolve.

L'importazione è gestita da `POST /avatars/import`. L'endpoint riceve un URL e applica una validazione preliminare della forma della URL (`http/https` con host presente), così da scartare input manifestamente non valido prima del download. Una volta accettato l'input, il server scarica l'asset in modo atomico: il download avviene su file temporaneo con estensione `.part` e viene rinominato al path finale solo al completamento, evitando che richieste concorrenti o interruzioni lascino in catalogo file parziali. Per la deduplicazione, il servizio calcola un hash SHA-256 della URL sorgente (`url_hash`); se un record con lo stesso hash è già presente e il file associato è risolvibile, l'import non replica il file ma riutilizza quello esistente, riducendo spazio su disco e tempi di rete. In parallelo, viene aggiornato un file di metadati JSON con operazioni anch'esse atomiche: la

scrittura avviene su un temporaneo e viene promossa tramite rename, così da garantire consistenza tra catalogo e file system anche in caso di crash.

Il serving del modello è esposto da `GET /avatars/{id}/model.glb`, che restituisce il contenuto tramite `FileResponse`. Prima di rispondere, il server risolve il path effettivo dell'asset con un meccanismo di *self-healing*: `resolve_avatar_file_path` controlla l'esistenza del file puntato dai metadati e, se non lo trova, tenta percorsi alternativi basati su convenzioni di directory e su nomi attesi. Questa strategia consente di recuperare da spostamenti o rinomini accidentali (ad esempio durante migrazioni o pulizie manuali), riducendo failure “hard” lato client e preservando l'utilizzabilità della libreria avatar senza richiedere una rigenerazione completa del catalogo. La gestione del ciclo di vita include anche `DELETE /avatars/{avatar_id}`, usato dal client quando un profilo viene rimosso.

Servizio	Endpoint	Input minimo	Output minimo / uso client
Whisper	POST /transcribe	form-data: file audio, language	JSON con text; usato per validazione campione vocale e turni vocali.
RAG	POST /chat	JSON: avatar_id, user_text, top_k	JSON con text, rag_used, auto_remembered; risposta conversazionale.
RAG	POST /ingest_file	form-data: avatar_id, file	JSON con ok, filename, chunks_added; ingest documenti in setup memoria.
RAG	POST /describe_image	form-data: avatar_id, remember, prompt, file	JSON con description e stato salvataggio; ingest semantico da immagine.
RAG	POST /remember	JSON: avatar_id, text, meta	JSON con ok, id; memorizzazione esplicita di note utente.
RAG	GET /avatar_stats; POST /clear_avatar	query/form-data: avatar_id, hard	JSON con stato memoria e reset soft/hard; controllo prerequisiti e cleanup.
Coqui TTS	POST /tts_stream	form-data: text, avatar_id, language, split_sentences, max_chunk_chars	stream audio/wav (header + chunk PCM16); playback incrementale.
Coqui TTS	POST /set_avatar_voice; GET/DELETE /avatar_voice	form-data o query: avatar_id, speaker_wav	JSON con esito, presenza e dimensione della voce; setup e reset riferimento vocale.
Coqui TTS	POST /generate_wait_phrases; GET /wait_phrase	form-data/query: avatar_id, language, name	JSON o audio/wav; pre-caching e riproduzione frasi d'attesa.
Avatar asset	GET /avatars/list	nessuno	JSON con avatars; popolamento libreria avatar nel frontend.
Avatar asset	POST /avatars/import	JSON: avatar_id, url, gender/bodyId/urlType	JSON con avatar_id, cached_glb_url; import con deduplica su url_hash.
Avatar asset	DELETE /avatars/{avatar_id}	path: avatar_id	JSON con ok, deleted; rimozione asset e metadati dal catalogo.

Tab. 4.1: Endpoint backend effettivamente invocati dal client Unity e contratto minimo di input/output.

4.3 Integrazione end-to-end

Mettere insieme servizi con latenze diverse (STT, RAG, TTS, asset) significa che ogni chiamata può fallire in modo diverso: timeout su Whisper, risposta RAG lenta, stream TTS interrotto o endpoint asset non raggiungibile. Lo stesso client deve inoltre operare sia in ambiente locale, dove le porte dei micro-servizi sono raggiungibili direttamente, sia in produzione, dove un reverse proxy fornisce un punto d'ingresso unico e riscrive i path. Orchestrazione delle richieste, normalizzazione degli URL e gestione esplicita dei failure servono a mantenere la UI coerente anche in presenza di errori transitori.

4.3.1 Orchestrazione richieste tra client, proxy e micro-servizi

Nel prototipo l'orchestrazione delle richieste è centralizzata in `UIFlowController.cs`, che coordina il ciclo conversazionale tipico: acquisizione audio in PTT, trascrizione, recupero contesto e generazione della risposta, sintesi vocale e riproduzione sincronizzata con il comportamento dell'avatar. In `MainMode` l'utente avvia la registrazione (tastiera o touch), il client produce un WAV e lo invia al servizio STT tramite `UnityWebRequest` (multipart form) verso l'endpoint `/transcribe`. Dopo la deserializzazione del JSON di risposta, il testo trascritto viene inoltrato al servizio RAG con una POST JSON verso `/chat`, includendo l'identificatore dell'avatar per selezionare la memoria per-profilo. La risposta testuale viene poi visualizzata e inviata al servizio TTS per la sintesi. Nel caso di sintesi streaming, il client riceve PCM progressivo, lo accoda e lo riproduce su `AudioSource`, mantenendo la UI aggiornata sugli stati *listening/processing/speaking*. Il lip-sync e gli altri controller comportamentali dell'avatar vengono attivati durante la riproduzione, così da rendere l'output coerente con la resa embodied descritta nelle sottosezioni precedenti.

Tutte le richieste HTTP sono configurate a partire da `SoulframeServicesConfig`, un `ScriptableObject` che centralizza i *base URL* (`whisperBaseUrl`, `ragBaseUrl`, `avatarAssetBaseUrl`, `coquiBaseUrl`) e una policy minima di invocazione (timeout e numero di retry). Questo accentramento evita URL hardcoded nel codice della UI e consente di propagare differenze di deploy senza modificare la logica del flusso. In ambiente di produzione WebGL,

il browser non contatta direttamente le porte interne 8001–8004, ma invia le richieste a un reverse proxy (ad esempio Nginx o Caddy) che espone un unico origin HTTPS e instrada i path `/api/whisper/*`, `/api/rag/*`, `/api/avatar/*`, `/api/tts/*` verso i corrispondenti servizi in loopback. Dal punto di vista architetturale questo nodo concentra routing e terminazione TLS e riduce la complessità lato client. L'effetto pratico è duplice: si preserva la separazione delle responsabilità dei servizi, ma si fornisce al client un accesso stabile e compatibile con i vincoli di CORS e stessa origine tipici di WebGL.

4.3.2 Normalizzazione endpoint locale vs produzione

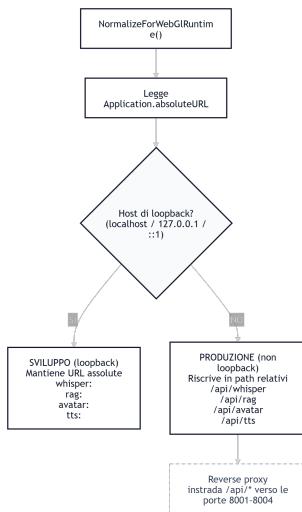


Fig. 4.9: Logica di normalizzazione degli endpoint in WebGL: se la pagina web è servita da loopback, si mantengono le URL assolute alle porte locali; altrimenti si usano path relativi per il routing tramite reverse proxy.

Il passaggio tra ambiente locale e produzione viene gestito in modo esplicito da `SoulframeServicesConfig.cs` tramite il metodo `NormalizeForWebGLRuntime()`. La funzione è attiva solo in build WebGL (`#if UNITY_WEBGL && !UNITY_EDITOR`) e introduce una regola semplice: se la pagina che ospita la build non è servita da un host di loopback, i base URL assoluti alle porte locali vengono riscritti in path relativi sotto `/api`. In particolare, si calcola `useRelativeApiPaths` come negazione di `IsCurrentWebPageLoopbackHost()`, e si normalizzano i servizi con `NormalizeServiceBaseUrl(...)` per ottenere rispettivamente `/api/whisper`, `/api/rag`, `/api/avatar` e `/api/tts`. In locale, quando la pagina è raggiunta da `localhost` o `127.0.0.1`, i base URL restano invece assoluti (`http://127.0.0.1:8001` ecc.), consentendo allo sviluppatore di testare ogni servizio direttamente sulla propria macchina.

La decisione operativa si basa su due controlli complementari: un helper verifica se la pagina è servita da un host loopback (`127.0.0.1`, `localhost`, `::1`), mentre un secondo

helper applica la riscrittura dei base URL verso i path `/api/*` solo quando necessario. La normalizzazione resta conservativa: in configurazioni non standard i valori non vengono riscritti.

La motivazione operativa di questa normalizzazione è evitare che il client WebGL, quando è servito su dominio pubblico, tenti di contattare `127.0.0.1:800x` dal browser dell'utente, indirizzo che in quel contesto punta alla macchina locale dell'utente e non al server. L'uso di path relativi sotto `/api` vincola invece le chiamate allo stesso origin che ha servito la build, delegando al reverse proxy il routing verso i micro-servizi interni. In questo modo si mantiene una singola configurazione nel client e si riducono errori dovuti a host hardcoded, differenze di porta e vincoli di sicurezza del browser.

4.3.3 Gestione errori, retry e fallback

L'integrazione end-to-end espone il client a failure tipici delle chiamate di rete (latenza variabile, timeout, servizi momentaneamente non raggiungibili) e richiede una gestione esplicita degli errori per evitare stati UI bloccanti. Nel prototipo questa gestione è concentrata in una policy semplice, definita in `SoulframeServicesConfig.cs`: `requestTimeoutSeconds` (default `15f`, con vincolo `[Min(1f)]`) stabilisce il tempo massimo atteso per una singola richiesta, mentre `retryCount` (default `1`, con vincolo `[Min(0)]`) definisce il numero massimo di ritentativi. La scelta di mantenere questi parametri in uno `ScriptableObject` consente di bilanciare reattività e tolleranza agli errori senza modificare il codice del flusso conversazionale.

Lato client C#/Unity, ogni `UnityWebRequest` imposta un `timeout` pari a `requestTimeoutSeconds`. Le coroutine che incapsulano le chiamate (ad esempio verso STT, RAG e TTS) valutano l'esito a partire dallo stato della request: un fallimento include timeout, errori di rete e risposte HTTP di classe 5xx. In tali casi il client ritenta l'invocazione fino a `retryCount` volte, ricostruendo la request e ripetendo l'invio. Questo schema non introduce backoff esponenziale o jitter, ma riduce i fallimenti transitori più comuni (ad esempio un primo tentativo perso per congestione o per cold start del servizio) mantenendo comunque una latenza massima prevedibile per l'utente, dato che il numero di retry è limitato. Quando anche i ritentativi falliscono, la UI aggiorna lo stato con un messaggio esplici-

to e termina la pipeline in modo pulito, ripristinando l’interazione (ad esempio tornando allo stato di ascolto o consentendo una nuova registrazione).

Per la fase TTS, in cui la latenza percepita è più critica, il fallback audio di cortesia è quello descritto in sottosezione 4.2.3. In questa sottosezione conta che il fallback mantenga feedback immediato anche quando la sintesi principale è in ritardo o richiede un retry.

Sul lato server, il servizio TTS implementa inoltre un fallback CUDA→CPU: quando `torch.cuda.is_available()` risulta falso, il modello XTTS viene caricato su CPU. L’esecuzione risulta più lenta, ma l’istanza continua a fornire sintesi senza interruzione del servizio, rendendo più stabile il deploy su macchine prive di GPU o su ambienti in cui la GPU non è disponibile temporaneamente.

Il livello di robustezza descritto è dimensionato per un prototipo di ricerca: `ai_services.cmd` non implementa health-check automatici né auto-ripristino, e le limitazioni che emergono in scenari prolungati sono discusse in 4.4.

4.4 Criticità affrontate e soluzioni

Nel percorso client–micro-servizi emergono criticità operative che in un prototipo monolitico restano spesso nascoste: latenza variabile tra stadi, vincoli browser sul routing API, qualità non uniforme dell’OCR, compatibilità fragile tra dipendenze ML e differenze di gestione tra Windows e Ubuntu. Le sottosezioni seguenti analizzano ogni area con lo stesso schema operativo: sintomo osservato, causa tecnica, soluzione adottata e impatto sul comportamento del sistema.

4.4.1 Latenza e timeout

Nel prototipo la criticità più immediata è la latenza percepita tra la fine dell’input vocale e l’inizio della risposta sintetizzata. In `MainMode` questo ritardo si manifesta come una sequenza di stati *listening*→*processing* che, se troppo lunga, viene interpretata dall’utente come un blocco dell’interazione. Il problema non dipende da una singola chiamata, ma dalla somma di più stadi: trascrizione Speech-to-Text, generazione contestuale con memoria, sintesi Text-to-Speech e avvio della riproduzione audio. Il caso peggiore coincide spesso con un *cold start*

(ad esempio modelli non ancora in cache o inizializzati), oppure con condizioni di carico in cui il runtime di inferenza impiega più tempo a restituire un risultato.

La causa tecnica è quindi compositiva. Il servizio STT carica un modello Whisper all'avvio e poi esegue l'inferenza su file audio caricati dal client; la fase di retrieval e chat del RAG dipende da chiamate a un runtime esterno (Ollama) che esegue embedding e Large Language Model (LLM); il servizio TTS usa un modello XTTS che, quando condizionato da un riferimento vocale per-avatar, può introdurre ulteriore costo computazionale, variando in modo sensibile tra esecuzione su GPU e su CPU. In presenza di questi tempi, una policy di rete troppo ottimistica produce fallimenti per timeout; una policy troppo permissiva rischia invece di mantenere la UI in attesa per intervalli poco predibili.

Per mitigare il problema si combinano misure lato client e lato server. Sul client, `SoulframeServicesConfig` centralizza una policy minima di invocazione con `requestTimeoutSeconds=15f` e `retryCount=1`; `UIFlowController` applica tali parametri impostando il timeout sulle `UnityWebRequest` e ripetendo la richiesta per un numero limitato di tentativi, così da assorbire failure transitori senza introdurre attese indefinite. In fase di tuning, il valore di 10 secondi (fallback usato in alcuni rami del client in assenza di config) si è mostrato troppo aggressivo su turni lenti, causando interruzione prematura del ciclo vocale; 15 secondi è risultato il valore pratico più adatto all'hardware di sviluppo. Sul server TTS, la latenza percepita viene ridotta tramite sintesi in streaming: l'endpoint `/tts_stream` emette un header WAV iniziale seguito da chunk PCM progressivi, permettendo al client di iniziare la riproduzione prima del completamento della sintesi complessiva. Per warmup/preload, fallback CUDA→CPU e fallback audio di cortesia si rimanda a sottosezione 4.2.3; qui conta che queste misure contengono il ritardo del primo turno e mantengono il servizio disponibile anche in assenza di GPU.

L'impatto pratico è un comportamento più robusto: si contengono i casi di UI bloccata e si stabilizza la latenza percepita, soprattutto per risposte lunghe o per i primi turni dopo l'avvio. Restano tuttavia limitazioni coerenti con la natura prototipale del sistema: la policy di retry è volutamente semplice (senza backoff esponenziale), e la pipeline continua a risentire dei tempi di inferenza quando i servizi sono avviati su hardware non adeguato o quando i modelli vengono eseguiti in modalità CPU.

4.4.2 CORS e routing API

In ambiente WebGL il browser può bloccare le chiamate del client verso i micro-servizi anche quando essi sono correttamente in esecuzione. Il problema si manifesta come errore lato frontend (tipicamente in console) e come assenza di traffico effettivo verso gli endpoint, perché l'intercettazione avviene prima che la richiesta raggiunga il server. La causa è la *Same-Origin Policy* (SOP), che isola le risorse tra origin diversi (schema, host e porta) e richiede un consenso esplicito per consentire accessi *cross-origin*. Il meccanismo standard per gestire tale consenso è il *Cross-Origin Resource Sharing* (CORS), che consente al server di dichiarare in modo controllato quali origin e metodi siano ammessi; la letteratura osserva però come design, implementazione e deployment di CORS risultino spesso soggetti a complessità e misconfigurazioni nel mondo reale [16].

Nel prototipo la condizione cross-origin emerge naturalmente dalla scomposizione a micro-servizi: in locale i servizi sono in ascolto su porte distinte (8001–8004), mentre la build WebGL può essere servita su un'altra porta o su un dominio differente. La criticità, quindi, non è un errore accidentale, bensì una conseguenza diretta della scelta architetturale.

Per risolvere il vincolo CORS si adottano due livelli complementari. Il primo è un'abilitazione CORS lato server: ciascuna applicazione FastAPI aggiunge un middleware `CORSMiddleware` con `allow_origins=["*"]` e `allow_methods=["*"]`, così da rendere possibili le chiamate cross-origin durante sviluppo e test, evitando che Unity/WebGL venga bloccato quando i servizi risiedono su porte diverse. Il secondo livello è la normalizzazione endpoint descritta in sottosezione 4.3.2, che in produzione elimina il cross-origin usando path relativi `/api/*` dietro reverse proxy.

L'impatto è duplice. In locale si preserva la possibilità di contattare direttamente `http://127.0.0.1:800x` per debugging e profiling dei singoli micro-servizi; in produzione si ottiene un unico origin HTTPS con routing stabile, riducendo i problemi di CORS e impedendo che il client tenti di raggiungere `127.0.0.1` dal browser dell'utente finale. La limitazione residua è che una policy CORS permissiva è adatta a un prototipo in rete controllata, mentre un rilascio pubblico richiederebbe una restrizione esplicita degli origin ammessi e una gestione più rigorosa delle policy lato proxy.

4.4.3 OCR e qualità dell'ingestione

Nel prototipo la qualità dell'ingestione nella memoria RAG dipende in modo critico dall'OCR (Optical Character Recognition) applicato a PDF e immagini. Il problema si osserva in modo concreto quando i documenti sono scansioni con risoluzione e contrasto variabili, font non standard o impaginazioni complesse (ad esempio multi-colonna e tavole): in questi casi il testo estratto contiene frammentazioni, inversioni di ordine e porzioni mancanti, che riducono l'efficacia del retrieval e aumentano la probabilità di recuperare contesto poco utile. La causa tecnica è che il servizio RAG tratta i PDF come immagini, rendendo esplicita la conversione pagina→bitmap e poi applicando un OCR generalista; questa scelta evita di affidarsi a testo embedded spesso corrotto, ma espone ai limiti tipici della trascrizione da immagini quando manca un pre-processing layout-aware.

La soluzione adottata in `rag_server.py` struttura l'ingestione come pipeline deterministica. Per i PDF, ogni pagina viene renderizzata con PyMuPDF (`fitz`) tramite `page.get_pixmap` a una risoluzione configurabile in DPI (Dots Per Inch), impostata di default a `RAG_OCR_DPI=400`, e convertita in PNG¹⁶. L'immagine viene poi passata a `pytesseract.image_to_string` con lingua configurabile `RAG_OCR_LANG=ita+eng`¹⁷, con un fallback su inglese se la configurazione non è disponibile. Il testo OCR viene normalizzato (`clean_text`) e linearizzato (`linearize_table_text`) per ridurre il rumore dovuto a newline e frammentazione tipica delle tavole, favorendo la similarità semantica negli embedding. L'indicizzazione applica poi un chunking per caratteri con overlap (`chunk_text` con `RAG_CHUNK_CHARS` e `RAG_CHUNK_OVERLAP`), scartando segmenti troppo corti e contenuti *garbage* tramite filtri conservativi; una deduplicazione intra-file basata su `seen_chunks` evita di inserire duplicati identici, mentre la deduplicazione per similarità viene usata in altri passaggi per ridurre ridondanze.

L'impatto sul prototipo è una memoria più pulita e interrogabile, perché il retrieval non viene saturato da frammenti vuoti o ripetitivi e l'estrazione OCR mantiene una copertura minima anche su documenti non nativamente testuali. Restano però limitazioni residue: la pipeline non implementa un'analisi avanzata

¹⁶ Doc ufficiale: PyMuPDF, <https://pymupdf.readthedocs.io/en/latest/>.

¹⁷ Doc ufficiale: Tesseract OCR, <https://tesseract-ocr.github.io/tessdoc/Installation.html>.

del layout (ad esempio detection di colonne o ricostruzione strutturata di tabelle), quindi l'ordine logico del contenuto può risultare imperfetto; inoltre la qualità dipende dal DPI di rendering e da caratteristiche grafiche difficili (font decorativi, scansioni inclinate). In questo contesto l'adozione di Tesseract come motore open-source multilingua risulta coerente con un approccio data-driven e generalista, che richiede poca personalizzazione oltre alla disponibilità delle risorse linguistiche, ma non elimina la dipendenza dalla qualità dell'immagine in input [17].

4.4.4 Compatibilità dipendenze/modelli

Un secondo problema ricorrente riguarda la compatibilità dell'ambiente Python necessario per eseguire i micro-servizi. In fase di installazione, errori di import o mismatch di ABI si manifestano in modo concreto come fallimenti all'avvio di Whisper, ChromaDB o, soprattutto, del servizio TTS. La causa tecnica è legata alla profondità delle dipendenze transitive nel dominio AI/ML: librerie come `coqui-tts`, `torch`, `transformers` e `tokenizers` impongono vincoli di versione stringenti e spesso dipendono da componenti compilati, rendendo fragile l'aggiornamento non coordinato dei pacchetti. Questo scenario è tipico nell'ecosistema PyPI, dove aggiornamenti upstream possono introdurre conflitti tra vincoli e ambiente locale, richiedendo strategie di risoluzione esplicite [18].

La soluzione adottata nel prototipo è un pinning rigoroso delle versioni in `requirements.txt`, che fissa l'accoppiata tra framework web e librerie ML (ad esempio `fastapi==0.128.0`, `openai-whisper==20250625`, `chromadb==1.4.1`, `coqui-tts==0.27.5`, `torch==2.10.0`, `torchaudio==2.10.0`, `transformers==4.57.1`, `tokenizers==0.22.1`, `torchcodec>=0.8.0`). Tale scelta riduce la variabilità tra macchine e rende riproducibile il comportamento durante test e deploy. Per le installazioni con accelerazione GPU, lo script operativo `ai_services.cmd` prevede inoltre un comando dedicato (`TORCH_INSTALL_CMD`) per installare wheel CUDA specifiche, evitando mismatch tra driver, build di PyTorch e librerie audio. Nel README del backend sono documentati anche casi di conflitto risolti: un errore del tipo `ImportError: cannot import name ... from transformers.pytorch_utils` viene mitigato riallineando `transformers==4.57.1` e `tokenizers==0.22.1`, mentre l'errore `torchcodec library is required for audio IO` viene risolto imponendo `torchcodec>=0.8.0`.

Questi esempi rendono esplicito che il problema non è solo teorico, ma emerge in modo ripetibile su ambienti reali.

L’impatto principale è un miglioramento della riproducibilità: un nuovo setup tende a convergere verso uno stato funzionante senza interventi ad hoc, e le regressioni possono essere tracciate a modifiche puntuali dei file dei requirements. Rimane tuttavia una fragilità strutturale rispetto a cambiamenti upstream: l’aggiornamento di un singolo pacchetto può richiedere riallineamenti a catena, e alcune dipendenze (in particolare nel TTS) impongono vincoli non sempre dichiarati in modo immediato nei metadati pubblici¹⁸. Nel perimetro attuale questo compromesso resta accettabile, ma suggerisce che un’evoluzione verso un rilascio più stabile richiederebbe una gestione più strutturata dell’ambiente (ad esempio lockfile e pipeline di build ripetibile) per ridurre ulteriormente l’impatto degli aggiornamenti remoti.

4.4.5 Differenze operative tra Windows e Ubuntu

Nel passaggio dall’ambiente di sviluppo Windows a quello di produzione Ubuntu, la differenza principale non è solo nel sistema operativo ma nel modello operativo. Su Windows il ciclo è centrato su `ai_services.cmd`, orientato a sviluppo locale rapido; su Ubuntu il flusso è stato separato in due ruoli esplicativi: `setup_soulframe_ubuntu.sh` per provisioning iniziale e `sf_admin_ubuntu.sh` per operazioni ricorrenti e aggiornamenti. Questa separazione evita che attività con obiettivi diversi (bootstrap infrastrutturale e manutenzione quotidiana) condividano lo stesso script.

La prima criticità riguarda la gestione dei processi. In sviluppo, `ai_services.cmd` avvia processi separati tramite `start` e in `stop/restart` usa `netstat+taskkill`; il meccanismo è adeguato per test locali ma non offre supervisione robusta nel lungo periodo. Su Ubuntu, `setup_soulframe_ubuntu.sh` genera unit `systemd` dedicate (`soulframe-whisper`, `soulframe-rag`, `soulframe-avatar`, `soulframe-tts`) con `Restart=always`, le aggrega in `soulframe.target` e installa il wrapper operativo `sfctl` per `start/stop/restart/status/logs` con diagnostica su `journalctl`. In questo modo la continuità dei servizi non dipende da sessioni terminali aperte e l’osservabilità resta uniforme tra riavvii e deploy successivi.

¹⁸ Doc ufficiale: PyPI coqui-tts, <https://pypi.org/project/coqui-tts/>.

Una seconda area è la riproducibilità dell'ambiente runtime. Lo script Ubuntu seleziona in modo deterministico il runtime Python (`python3.12` → `python3.11` → `python3.10` → `python3`), installa i pacchetti di sistema necessari via `apt` (inclusi `tesseract-ocr-ita/eng`) e crea un venv stabile in `/opt/soulframe/.venv`, con configurazione centralizzata in `/etc/soulframe/soulframe.env`. Il setup include anche fix automatici su conflitti noti (`transformers/tokenizers`, `torchcodec`) e supporta un override GPU tramite `TORCH_INSTALL_CMD`. Il risultato è una convergenza più affidabile tra macchine diverse, al costo di una minore libertà di modifica estemporanea rispetto al flusso Windows.

Sul fronte rete, in locale resta utile il controllo diretto degli endpoint `127.0.0.1:800x`, ma in produzione Ubuntu lo script genera un `Caddyfile` con routing `/api/*` verso i micro-servizi interni e con un unico origin HTTPS pubblico. Questa configurazione riduce i problemi CORS in WebGL, mantiene allineata la normalizzazione endpoint lato client e concentra il controllo del traffico su un nodo centrale. La scelta è motivata da robustezza operativa: meno configurazioni duplicate nel frontend e minore rischio di host hardcoded non validi nel contesto browser.

Un ulteriore nodo operativo è l'aggiornamento. `sf_admin_ubuntu.sh` usa una cartella di drop (`soulframe_update`, esposta anche via `SOULFRAME_UPDATE_DIR`), rileva automaticamente artefatti supportati (backend, script, `Build.zip`), applica backup versionati e normalizza i file aggiornati (LF/perms) prima del rilancio servizi. Quando vengono aggiornati gli script di setup/admin, il flusso reinstalla `sfadmin` e rilancia il setup in modalità controllata (`SKIP_OLLAMA_PULL=1`) per evitare side effect inutili. Questa catena rende esplicito il trade-off accettato: operazioni più disciplinate e tracciabili, con la necessità di privilegi amministrativi (`sudo/root`) tipica di un ambiente server.

4.5 Runbook operativo essenziale

Nel runbook, avvio, verifica, aggiornamento e manutenzione ordinaria sono formalizzati con procedure ripetibili. I controlli restano essenziali: riconoscere rapidamente un servizio non in salute, distinguere un problema di routing da un problema applicativo, applicare un aggiornamento senza interrompere stabilmente la piattaforma e mantenere traccia delle modifiche tramite backup. La distinzione

tra ambiente di produzione Ubuntu e sviluppo locale Windows resta esplicita, perché i due contesti impongono strumenti operativi diversi pur condividendo le stesse porte logiche e lo stesso set di servizi.

Provisioning dell'ambiente di produzione (Ubuntu). Il provisioning in produzione è automatizzato dallo script `setup_soulframe_ubuntu.sh`, che prepara un layout coerente sotto `/opt/soulframe` e separa i parametri di runtime in `/etc/soulframe/soulframe.env`. Sul piano operativo, la scelta principale è trattare ciascun micro-servizio come un'unit `systemd` dedicata: `soulframe-whisper.service` (porta 8001), `soulframe-rag.service` (8002), `soulframe-avatar.service` (8003), `soulframe-tts.service` (8004), affiancate da `soulframe-ollama.service` come wrapper di controllo per Ollama e da `soulframe.target` come aggregatore per la gestione coordinata. Ogni unit configura `WorkingDirectory=/opt/soulframe/backend` e avvia il relativo server Fa-stAPI tramite `uvicorn` in loopback (`-host 127.0.0.1`), limitando l'esposizione diretta delle porte al solo host. La robustezza in esercizio viene ottenuta con direttive di restart automatico (`Restart=always`, `RestartSec=5`) e con timeout di start dimensionati in base al costo di inizializzazione dei modelli, demandando a `systemd` il recupero da fault transitori senza richiedere interventi manuali continui.¹⁹

```

1 [Service]
2 Type=simple
3 WorkingDirectory=/opt/soulframe/backend
4 EnvironmentFile=/etc/soulframe/soulframe.env
5 ExecStart=/opt/soulframe/.venv/bin/uvicorn whisper_server:app --host
    127.0.0.1 --port 8001
6 Restart=always
7 RestartSec=5
8 TimeoutStartSec=180

```

In parallelo, lo stesso provisioning genera un `Caddyfile` che svolge due ruoli operativi: serve la build WebGL come sito statico e funge da reverse proxy

¹⁹ `systemd.service` (Restart, RestartSec, ExecStart, WantedBy) — <https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html>

unico per le API. Il routing è organizzato per prefissi `/api: /api/whisper/*` inoltra a `127.0.0.1:8001`, `/api/rag/*` a `127.0.0.1:8002`, `/api/avatar/*` a `127.0.0.1:8003` e `/api/tts/*` a `127.0.0.1:8004`. Dal punto di vista operativo questo consente terminazione TLS e origin unico HTTPS verso il browser, coerentemente con la logica di riscrittura dei base URL descritta in sottosezione 4.3.2.²⁰ Lo script installa inoltre helper CLI in `/usr/local/bin`: `sfctl` per la gestione dei servizi, `sfurl` per la stampa degli URL di health-check e `sfadmin` come entrypoint alla console amministrativa `sf_admin_ubuntu.sh`. Come misura accessoria di contenimento costi, viene configurato anche un controllo periodico di inattività con `idle_shutdown.sh` e relativo timer, che può spegnere la VM dopo una soglia di idle basata su accessi recenti alle API e, opzionalmente, su attività SSH.

Gestione quotidiana (`sfctl`, `sfurl`, `log`). Una volta completato il provisioning, la gestione quotidiana privilegia comandi idempotenti e verifiche a basso costo cognitivo. L’interfaccia primaria è `sfctl`, che incapsula `systemctl` su singolo servizio o su tutti i servizi tramite il target `soulframe.target`. In avvio, `sfctl start` attiva l’intero gruppo; in stop, `sfctl stop` arresta esplicitamente le unit per evitare che il solo stop del target lasci componenti attivi; in restart, `sfctl restart` ripristina una condizione nota dopo cambi di configurazione o aggiornamenti. La verifica dello stato è ottenuta con `sfctl status`, che stampa lo stato delle unit principali senza paginazione; per l’analisi dei log la procedura standard è `sfctl logs <servizio>` oppure `sfctl logs all`, che esegue un tail in follow tramite `journalctl -u <unit> -f` sugli stessi identificativi di unit e, nel caso aggregato, include anche i log del demone Ollama.²¹ Questa separazione è operativamente utile: un errore di rete o di timeout lato client ha spesso correlazione con un riavvio del servizio o con un errore applicativo specifico, mentre un problema di routing tende a manifestarsi come assenza di richieste in ingresso sul servizio bersaglio.

²⁰ Caddy (reverse_proxy) — https://caddyserver.com/docs/caddyfile/directives/reverse_proxy

²¹ journalctl (filtri per unit e follow) — <https://www.freedesktop.org/software/systemd/man/latest/journalctl.html>

La seconda verifica, complementare ai log, è basata su health-check HTTP. `sfurl` legge il dominio configurato in `/etc/soulframe/soulframe.env` e stampa gli endpoint `/api/*/health` esposti dal reverse proxy, rendendo immediato distinguere un fault di applicazione (risposta `/health` anomala o assente) da un fault di origin o di certificato (impossibilità di raggiungere l'URL HTTPS). In contesti headless, `sfurl` resta utile anche senza apertura automatica del browser, perché espone in forma testuale lo stesso insieme minimo di URL che il frontend WebGL utilizzerà in produzione. In caso di diagnosi più fine, i servizi FastAPI espongono anche la UI di documentazione automatica (`/docs`); tuttavia nel runbook si privilegia `/health` perché è un controllo leggero e adatto a essere automatizzato o eseguito frequentemente senza rumore.

Pipeline di aggiornamento e manutenzione (sfadmin). L'aggiornamento in produzione è progettato come sequenza unica e ripetibile, per ridurre interventi manuali su cartelle di deploy e minimizzare errori di permessi o copie parziali. La console `sfadmin` (`sf_admin_ubuntu.sh`) viene eseguita con privilegi elevati e centralizza le azioni in un menu che include start/stop/restart, modifica parametri e update. L'update unificato (`update_all`) opera su una directory di drop (`UPDATE_DIR`) che viene risolta con priorità deterministica: variabile d'ambiente `UPDATE_DIR` se impostata, altrimenti `SOULFRAME_UPDATE_DIR` in `/etc/soulframe/soulframe.env`, altrimenti fallback nella home dell'utente (`.../soulframe_update`). La scelta di una cartella di drop separata dal deploy evita di modificare direttamente `/opt/soulframe` con file intermedi o incompleti e rende più semplice trasferire artefatti via `scp` o strumenti analoghi.

Dal punto di vista delle operazioni, `update_all` arresta i servizi, aggiorna la build WebGL da un archivio ZIP (preferendo esplicitamente `Build.zip`, altrimenti l'ultimo `*.zip` disponibile) e aggiorna backend e script a partire da un set ristretto di nomi file supportati. L'aggiornamento della build utilizza un estrattore in una directory temporanea e, se esiste già una build in `/opt/soulframe/webgl`, crea un backup `webgl_backup_YYYYMMDD_HHMMSS` prima di sincronizzare i nuovi contenuti (con `rsync -delete` quando disponibile). L'aggiornamento del backend applica un modello analogo: prima crea una cartella di backup `/opt/soulframe/backups/backend_update_YYYYMMDD_HHMMSS`, poi copia i file aggiornati nel target corretto con `install -m`, preservando permessi

coerenti e normalizzando i line ending (CRLF → LF) per `.sh` e `.py`. Se l’aggiornamento include `setup_soulframe_ubuntu.sh` e/o `sf_admin_ubuntu.sh`, la procedura reinstalla `sfadmin` e rilancia il setup con `SKIP_OLLAMA_PULL=1`, mantenendo l’operazione idempotente e limitando i costi di rete. A valle dell’update, la console propone una pulizia esplicita dei file sorgente usati, con conferma interattiva e vincolo di sicurezza: vengono rimossi solo file interni a `UPDATE_DIR`, evitando cancellazioni accidentali fuori dalla cartella di drop.

Il ciclo di vita operativo, dalla verifica di stato al restart e all’update, è riassunto in Figura 4.10. La figura non introduce nuovi dettagli implementativi, ma visualizza la sequenza tipica con cui si passa da una diagnosi (`status/log/health`) a una correzione (`restart` o `update`) e al successivo controllo di corretto ripristino.

Ambiente di sviluppo locale su Windows (ai_services.cmd). In ambiente Windows, il runbook adotta una logica orientata allo sviluppo locale, dove l’obiettivo primario è ridurre l’attrito di avvio e mantenere visibili i log di ogni componente. Lo script `ai_services.cmd` fornisce comandi `start/stop/restart` (anche tramite menu interattivo) e avvia i servizi su porte locali senza reverse proxy: Whisper su 8001, RAG su 8002, Avatar Asset su 8003, TTS su 8004, più Ollama su 11434. Prima dell’avvio, una funzione di controllo (`PORT_IS_LISTENING`) verifica se una porta è già in ascolto e, in tal caso, evita di avviare duplicati; nella procedura di stop, una funzione dedicata (`KILL_PORT_PY`) termina selettivamente processi `python.exe` associati alla porta, limitando gli effetti collaterali su altre applicazioni. Per rendere pratico il test del client WebGL in locale, lo stesso script avvia anche un semplice server HTTP che serve la build sulla porta 8000, allineandosi al caso in cui il browser raggiunge `http://localhost:8000` e contatta direttamente `http://127.0.0.1:8001-8004`. La logica di normalizzazione locale-vs-produzione resta quella descritta in sottosezione 4.3.2.

Nel workflow locale è disponibile anche `SoulframeControlCenter.bat` come launcher operativo. Il tool inoltra i comandi `s/c/r` a `ai_services.cmd` e automatizza la preparazione del pacchetto `soulframe_update` (creazione `Build.zip` e copia di script/backend necessari al deploy Ubuntu). In pratica riduce operazioni ripetitive lato Windows, mentre in Ubuntu la gestione resta affidata a `systemd` + `sfctl/sfadmin`.

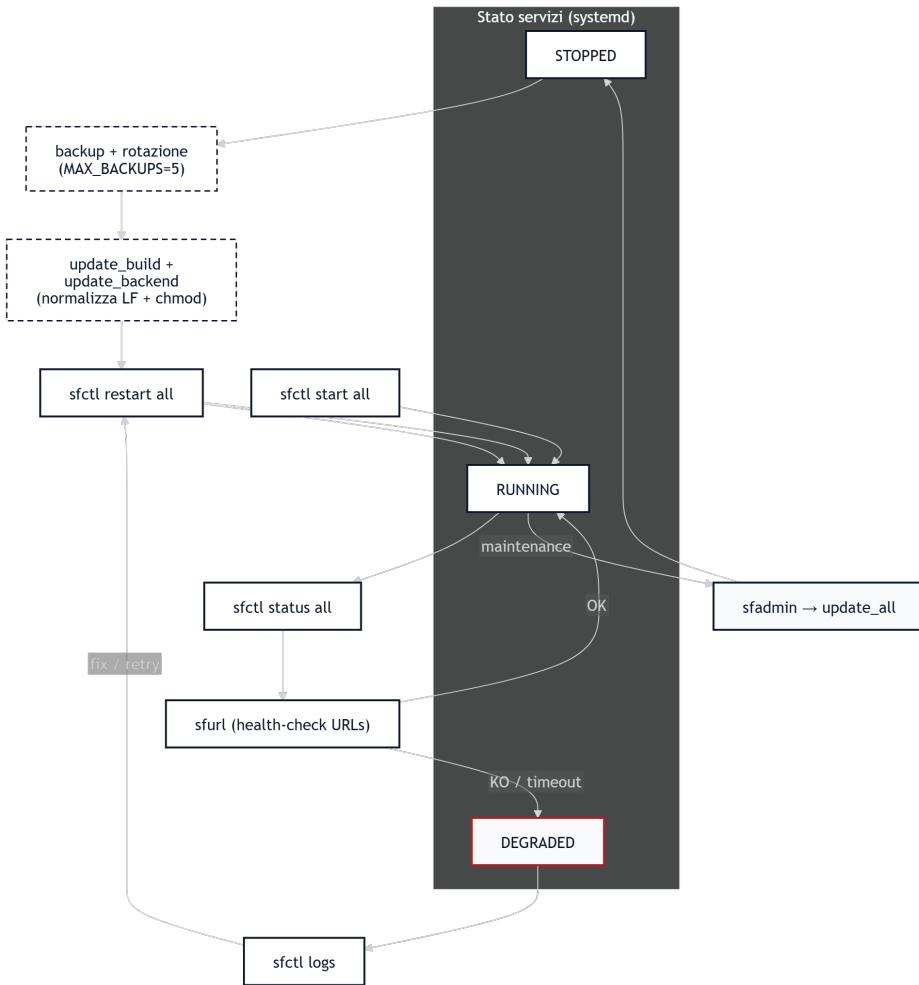


Fig. 4.10: Ciclo di vita operativo dei servizi SOULFRAME gestito dagli script `sfctl` e `sf_admin_ubuntu.sh`.

Chiusura: garanzie operative e limiti del runbook. Il runbook qui descritto rende esplicite le operazioni minime necessarie a mantenere il prototipo in uno stato utilizzabile, ma non sostituisce meccanismi di hardening tipici di sistemi in produzione. La resilienza è demandata principalmente a `systemd` (restart automatici) e a una disciplina di update basata su backup e sincronizzazione atomica degli artefatti; non sono presenti health-check automatici che disabilitino un servizio in caso di risposte degradate, né procedure di rollback automatizzate oltre alla disponibilità delle cartelle di backup create durante l'update. La combinazione di target `systemd`, reverse proxy e script amministrativi copre comunque i failure

più comuni emersi nelle sezioni precedenti: servizi non avviati, porte occupate, configurazioni incoerenti tra locale e produzione e aggiornamenti applicati in modo incompleto. In questo quadro la formalizzazione di poche procedure ripetibili ha un impatto pratico: il tempo speso nella diagnosi diminuisce, e le modifiche necessarie per iterare sul sistema restano tracciabili e riproducibili anche a distanza di tempo.

Mini runbook d'emergenza. In caso di servizio non disponibile, la sequenza minima è: (i) verificare stato unit con `sfctl status`, (ii) controllare l'endpoint `/api/*/health` con `sfurl`, (iii) correlare i log del servizio interessato con `sfctl logs <servizio>`, (iv) eseguire `sfctl restart <servizio>` o `sfctl restart` per ripristino coordinato. In caso di errore CORS, il controllo prioritario è verificare che il frontend stia usando path relativi `/api/*` in produzione e che il reverse proxy instradì correttamente verso `127.0.0.1:8001–8004` (coerenza con sottosezione 4.3.2). In caso di timeout TTS, la diagnosi minima è distinguere fault di rete da latenza di inferenza: se `/api/tts/health` è raggiungibile ma la risposta è lenta, il fallback audio di cortesia mantiene la continuità del turno mentre si verifica l'eventuale degradazione CUDA→CPU descritta in sottosezione 4.2.3.

4.6 Affidabilità e sicurezza operativa

In questa sezione rendo esplicito il perimetro: non stavo costruendo un sistema pronto per produzione, ma un prototipo funzionante e osservabile end-to-end. Ho quindi privilegiato scelte operative che riducono fault transitori e rendono la diagnosi ripetibile, accettando che alcune misure di sicurezza avanzata restassero fuori scope. Il punto di partenza è il runbook operativo (sezione 4.5), con procedure ripetibili per avvio, verifica, aggiornamento e diagnostica. L'architettura a micro-servizi introduce più punti di failure e più endpoint rispetto a un'applicazione monolitica, quindi richiede policy minime di timeout/retry (sottosezione 4.3.3), health-check praticabili e aggiornamenti che limitino il downtime. Sul fronte sicurezza, normalizzazione degli endpoint in WebGL (sottosezione 4.3.2) e reverse proxy con TLS definiscono un perimetro unico di ingresso.

Affidabilità: auto-restart, health-check e gestione dei fault. In ambiente di produzione Ubuntu, l'affidabilità operativa si fonda sulla delega della supervisione dei processi a `systemd`. Le unit generate dallo script di setup impostano `Restart=always` e `RestartSec=5`, così che un crash di un micro-servizio (Whisper, RAG, Avatar Asset o TTS) venga seguito da un riavvio automatico dopo cinque secondi.²² Questa scelta ha due effetti concreti: riduce la necessità di intervento manuale per fault transitori e limita la durata delle interruzioni dovute a errori non deterministici (ad esempio eccezioni runtime, problemi temporanei di I/O). Lo stesso provisioning definisce inoltre un `soulframe.target` che raggruppa le unit e specifica dipendenze e ordine di avvio tramite `Requires=` e `After=`, rendendo più prevedibile il ripristino dello stack quando l'intero set di servizi viene avviato o riavviato in modo coordinato.

Il meccanismo di auto-restart non sostituisce un health management completo, perché `systemd` reagisce al crash del processo ma non interpreta la qualità del servizio quando il processo resta vivo ma non risponde correttamente. Nel prototipo, i server FastAPI espongono endpoint di health-check (`/health`) utilizzati a scopo diagnostico dal runbook, in particolare tramite `sfurl` e `sfctl logs` per correlare un errore lato client con un evento lato server. In letteratura, health-check più strutturati includono tipicamente probe di *liveness* e *readiness* e si integrano con orchestratori o bilanciatori per isolare istanze non pronte o non sane, riducendo failure propagati e downtime; inoltre pattern come il *circuit breaker* consentono di contenere fault a cascata quando dipendenze esterne diventano intermittenti [19]. Nel contesto SOULFRAME, l'assenza di un orchestratore (ad esempio Kubernetes) implica che tali probe non siano sfruttati in modo automatico; tuttavia la presenza di `/health` e la disponibilità dei log centralizzati costituiscono un primo livello di osservabilità coerente con gli obiettivi di prototipazione.

La resilienza minima lato client (timeout e retry in `SoulframeServicesConfig`) è descritta in dettaglio in sottosezione 4.3.3. In questa sede conta l'effetto operativo complessivo: contenere i failure transitori nella catena STT→RAG→TTS mantenendo una latenza massima prevedibile.

²² `systemd.service` (Restart, RestartSec) — <https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html>

Affidabilità: aggiornamenti, rollback manuale e minimizzazione del downtime. La seconda componente di affidabilità operativa riguarda gli aggiornamenti, che rappresentano una fonte frequente di regressioni e downtime se applicati in modo parziale. Nel prototipo, la console amministrativa `sf_admin_ubuntu.sh` implementa una pipeline `update_all` che segue una sequenza deterministica: stop dei servizi, creazione di backup, deploy degli artefatti (backend e build WebGL), normalizzazione dei file e riavvio. Prima di ogni deploy, la funzione `manage_backups` applica una rotazione con `MAX_BACKUPS=5`, limitando crescita non controllata dello storage e mantenendo una finestra di ripristino ragionevole. L'aggiornamento, pur non essendo *zero-downtime* in senso stretto, minimizza la finestra di indisponibilità perché arresta i servizi solo durante la fase di copia e ripristina rapidamente lo stack; inoltre evita stati intermedi difficili da diagnosticare (ad esempio un backend aggiornato con frontend vecchio, o viceversa) perché tratta backend e WebGL come un'unità di rilascio.

Il rollback resta manuale: il sistema conserva i backup, ma non esegue automaticamente un revert in caso di failure post-deploy. Questa limitazione deriva dalla scelta di mantenere l'infrastruttura leggera; in un'evoluzione verso un rilascio più stabile, l'introduzione di verifiche automatiche post-deploy (ad esempio health-check su tutte le unit e test di richiesta minima STT→TTS) consentirebbe di bloccare o annullare automaticamente un aggiornamento non sano, avvicinando la pipeline a pratiche di continuous delivery più robuste.

Sicurezza operativa: isolamento di rete e perimetro TLS. La misura di sicurezza operativa più rilevante già presente è la riduzione della superficie di attacco tramite isolamento di rete. In produzione, tutti i micro-servizi FastAPI eseguono bind su `127.0.0.1`, quindi non sono direttamente esposti alla rete pubblica; l'unico punto di ingresso è il reverse proxy Caddy, che termina TLS e inoltra le richieste ai servizi interni tramite routing su `/api/*`. Il risultato è un perimetro unico: browser e utenti esterni comunicano solo con Caddy in HTTPS, mentre le porte 8001–8004 restano confinate al loopback del server. Caddy gestisce inoltre *automatic HTTPS*, includendo emissione e rinnovo dei certificati e riducendo errori operativi legati alla gestione manuale di TLS.²³ La direttiva `reverse_proxy`

²³ Caddy (Automatic HTTPS) — <https://caddyserver.com/docs/automatic-https>

definisce l'inoltro verso i target in loopback e consente di mantenere invariata la configurazione del client WebGL, che in produzione utilizza path relativi anziché URL assolute, come discusso in sottosezione 4.3.2.²⁴

Questo assetto fornisce un livello di sicurezza pragmatico per un prototipo: il traffico è cifrato sul tratto pubblico, e i servizi non sono raggiungibili direttamente dall'esterno. Restano però assenti altre misure di enforcement al perimetro: non è implementata autenticazione sugli endpoint, non è presente rate-limiting e le policy CORS risultano permissive (`allow_origins=["*"]` sui server FastAPI). In termini di threat model, ciò implica che chiunque possa raggiungere l'origin pubblico possa invocare le API, soggetto solo alla conoscenza degli endpoint e alla disponibilità dei servizi. Tale scelta è compatibile con un contesto controllato (ambiente di test o dimostrazione), ma non è adeguata a un'esposizione pubblica non supervisionata.

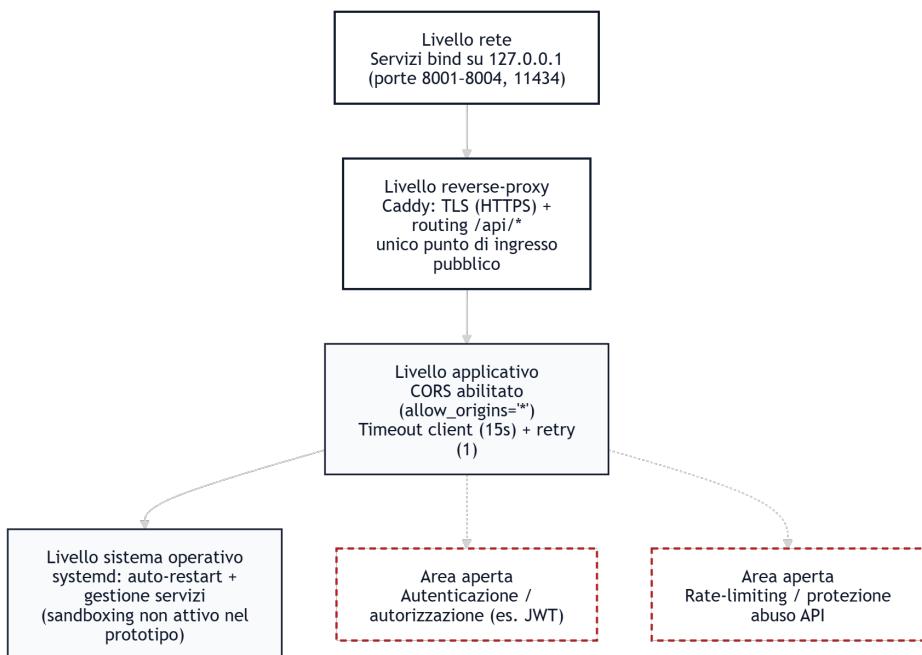


Fig. 4.11: Livelli di sicurezza dell'architettura SOULFRAME: rete, reverse-proxy, applicazione, sistema operativo.

L'organizzazione a livelli illustrata in Figura 4.11 chiarisce che, allo stato at-

²⁴ Caddy (reverse_proxy) — https://caddyserver.com/docs/caddyfile/directives/reverse_proxy

tuale, le misure operative si concentrano principalmente sui livelli rete e reverse proxy (bind locale e TLS), mentre il livello applicativo resta volutamente aperto per massimizzare la sperimentabilità della pipeline conversazionale. Questa impostazione riduce complessità e attrito durante lo sviluppo, ma sposta l'onere della sicurezza verso il contesto operativo in cui il prototipo viene eseguito.

Limiti attuali e possibili evoluzioni. La principale limitazione di sicurezza è l'assenza di autenticazione e autorizzazione centralizzate. In letteratura, l'API gateway viene frequentemente proposto come punto di enforcement per micro-servizi, delegando al gateway la verifica delle credenziali e l'applicazione di policy uniformi (ad esempio basate su JSON Web Token, JWT), così che i servizi interni possano mantenersi leggeri e focalizzati sulla logica di dominio [20]. In un'evoluzione di SOULFRAME, tale modello potrebbe essere implementato a livello Caddy o tramite un gateway dedicato, introducendo un controllo di accesso per le rotte `/api/*` e riducendo l'esposizione degli endpoint più sensibili (ad esempio upload audio per STT e upload reference per TTS). In modo complementare, un framework di mitigazione a livelli per API REST suggerisce di combinare autenticazione forte, validazione degli input, rate-limiting e centralizzazione delle policy al gateway per ridurre vettori comuni quali abuso di risorse, injection e accesso non autorizzato [21]. Nel prototipo, la validazione dei payload è delegata in parte a FastAPI/Pydantic, ma mancano limiti di traffico e controlli di identità lato server.

Sul versante affidabilità, l'assenza di un service mesh o di un orchestratore limita l'automazione dei health-check e delle strategie di contenimento dei fault. L'esperienza operativa mostra che il solo auto-restart è efficace contro crash, ma meno contro degradazioni silenziose. L'adozione di probe di *readiness* (per evitare di instradare traffico verso servizi non pronti, ad esempio durante warmup del TTS) e di pattern di *circuit breaker* (per limitare ritentativi verso dipendenze non disponibili e prevenire cascata di timeout) rappresenterebbe un'estensione diretta delle pratiche discusse in letteratura su tolleranza ai fault in architetture a micro-servizi [19]. Una prima estensione, senza cambiare piattaforma, può introdurre un controllo periodico dei `/health` a livello di reverse proxy o di script amministrativi, con restart automatico quando un servizio non risponde entro una soglia. Un'evoluzione più invasiva ma più robusta sarebbe migrare la gestione

dei servizi a un orchestratore che integri nativamente liveness/readiness e rolling update, riducendo la finestra di downtime durante aggiornamenti e abilitando rollback automatici.

L'implementazione corrente offre un nucleo solido per affidabilità operativa in un contesto prototipale: auto-restart dei servizi, isolamento di rete tramite bind locale, TLS automatizzato e procedure di update con backup e ripristino rapido. Ho mantenuto questo equilibrio perché era coerente con l'obiettivo della tesi: validare la pipeline conversazionale end-to-end con un'infrastruttura stabile, senza aprire in parallelo un cantiere completo di hardening da produzione. Il runbook (sezione 4.5) e la gestione degli errori (sottosezione 4.3.3) restano la base su cui innestare, in lavori successivi, controlli di accesso, rate-limiting e health management più automatico.

5. RISULTATI E VALUTAZIONE

5.1 *Impostazione della valutazione*

5.1.1 *Scenari di prova e setup sperimentale*

Le osservazioni sono state raccolte senza anticipare i risultati quantitativi: per un agente embodied vocale contano soprattutto regolarità del turno conversazionale e coerenza tra stato interno e feedback dell’interfaccia. Il quadro architettonale di riferimento è descritto nel Capitolo 3, mentre i dettagli implementativi della pipeline e del routing locale/server sono nel Capitolo 4 (sottosezione 4.3.1, sottosezione 4.3.2).

Le prove sono state svolte in due ambienti funzionalmente equivalenti. In locale, il client Unity in `MainMode` comunica con i micro-servizi su loopback (`127.0.0.1:8001-8004`). Su server (Ubuntu), la stessa pipeline è esposta dietro reverse proxy con path `/api/*`. La normalizzazione degli endpoint lato client mantiene invariata la logica applicativa e limita le differenze alla configurazione di rete.

L’ambiente locale Windows è un laptop con CPU AMD Ryzen AI 9 HX 370 e dGPU NVIDIA RTX 5070 Ti Mobile con 12 GB di VRAM. L’ambiente server Ubuntu testato è su Google Cloud, con GPU NVIDIA L4 (VRAM tipica 24 GB), disco persistente da 60 GB, IPv4 statico e dominio dedicato ai test. In questa configurazione, lo STT usa Whisper `small` come default in locale Windows e Whisper `medium` nel setup Ubuntu testato, coerentemente con la diversa disponibilità di VRAM.

Il flusso osservato segue il percorso operativo del prototipo: `POST /transcribe` per la trascrizione, `POST /chat` per retrieval e generazione, `POST /tts_stream` per la risposta vocale in streaming. Il confronto locale/server è stato condotto su sessioni omogenee per avatar, lingua e stato memoria (presente/assente), così da

limitare variabili confondenti.

I punti di misura sono stati fissati in modo uniforme (fine registrazione, completamento STT, completamento RAG, primo audio TTS, inizio playback). In questa versione le misure derivano da osservazione controllata e diagnostica di esecuzione (stato UI, esiti HTTP, log dei servizi), non da una pipeline di telemetria con persistenza automatica per turno. I risultati tecnici vanno quindi letti come evidenza sperimentale su campione ridotto di sessioni/turni: sono utili per confrontare configurazioni, ma non costituiscono ancora un benchmark statistico definitivo.

Gli scenari conversazionali coprono sia il percorso nominale sia i casi più sensibili alla latenza percepita: domanda aperta, interrogazione di memoria per-avatar, sequenze brevi e ravvicinate di turni, sessioni prolungate. Prima di ogni sessione è stata verificata la disponibilità dei servizi tramite endpoint `/health`, per ridurre fallimenti legati a boot incompleto o configurazioni incoerenti con lo scenario.

5.1.2 Metriche tecniche adottate

Le metriche selezionate sono coerenti con i requisiti non funzionali discussi in sezione 3.1 e con le politiche di resilienza applicate nel client e nei servizi (sottosezione 4.3.3). La valutazione tecnica misura reattività della pipeline vocale e tenuta del turno conversazionale. Sono usate tre definizioni operative: tempo al primo audio della fase TTS (tempo tra invio della richiesta `/tts_stream` e primo chunk audio riproducibile), latenza end-to-end del turno (tempo tra rilascio del push-to-talk e primo audio riprodotto) e *cold-start/time-to-ready* (tempo tra avvio dello stack servizi e stato “pronto all’interazione”).

Nella versione corrente risultano misurati in modo osservativo il tempo al primo audio della fase TTS, i tempi STT e il cold-start. Il tempo per-fase di RAG/LLM e la latenza end-to-end non sono invece misurati sistematicamente e sono riportati come n.d. Le latenze per fase servono ad attribuire il costo ai singoli stadi, mentre la latenza end-to-end descrive la latenza percepita di turno.

Per la robustezza operativa si osservano richieste fallite, timeout lato client e necessità di ripetere il turno. In coerenza con la configurazione corrente del client (`requestTimeoutSeconds=15, retryCount=1`), questa misura indica quanto spesso la pipeline si interrompe o degrada durante l’uso continuativo.

Per la qualità STT si adotta la Word Error Rate (WER), confrontando trascrizione Whisper e riferimento pronunciato. Le osservazioni considerano il profilo `small` in locale Windows e `medium` nel setup Ubuntu testato. Per la qualità RAG si distinguono retrieval quality e generation quality, analizzando pertinenza del contesto recuperato e fedeltà della risposta rispetto ai contenuti effettivamente usati (campo `rag_used`). L'impostazione segue la letteratura recente su valutazione RAG e mitigazione delle allucinazioni [22], oltre alla valutazione consolidata dei sistemi ASR [7].

5.1.3 Metriche di esperienza utente

A livello teorico, la scelta degli indicatori richiama il quadro di letteratura su ECA/Voice UX presentato nel Capitolo 2, in particolare sezione 2.1 e sezione 2.2. La valutazione UX integra osservazioni qualitative e una cornice standard di usabilità. In SOULFRAME la Voice UX dipende da tre fattori principali: chiarezza del turn-taking, comprensibilità del parlato sintetizzato e leggibilità dello stato conversazionale in interfaccia.

Le osservazioni qualitative sono organizzate su indicatori esplicativi: frequenza di ripetizioni/riformulazioni, numero di turni di riparazione, tolleranza all'attesa, stabilità del ritmo conversazionale, coerenza percepita tra voce e avatar. Per il sottosistema visivo si osservano anche chiarezza degli stati (listening/processing/speaking), comfort visivo (flicker, intensità bloom, movimenti camera) e coerenza stilistica complessiva.

Il target di valutazione resta ampio e non è ristretto a una singola fascia anagrafica. In questa fase, gli accorgimenti di accessibilità (icone dinamiche, feedback testuali e segnali visivi persistenti) sono trattati come requisiti trasversali di leggibilità dell'interazione, non come adattamenti per un unico profilo utente.

5.2 Risultati tecnici del prototipo

5.2.1 Prestazioni della pipeline STT-RAG-TTS

I risultati tecnici mostrano che la pipeline STT–RAG–TTS resta utilizzabile in entrambi gli ambienti testati. I valori riportati sono ordini di grandezza osservati su un campione ridotto di sessioni/turni esplorativi controllati, non misure da

Fase	Locale (Windows)	Server (Ubuntu)
STT (Whisper: small/medium)	nell'ordine di 1–3 s	nell'ordine di 1–4 s
RAG+LLM (Ollama)	non misurato sistematicamente (n.d.)	non misurato sistematicamente (n.d.)
TTS streaming (tempo al primo audio)	circa 5 s (tempo al primo audio)	circa 7–8 s (tempo al primo audio)
End-to-end (rilascio PTT → primo audio)	non misurato sistematicamente (n.d.)	non misurato sistematicamente (n.d.)

Tab. 5.1: Confronto indicativo dei tempi per fase della pipeline tra ambiente locale e server. La riga TTS riporta il tempo al primo audio; la latenza end-to-end è n.d.

campagna benchmark automatizzata. Le scelte implementative di riferimento sono nel Capitolo 4, soprattutto per i moduli RAG e TTS (sottosezione 4.2.2, sottosezione 4.2.3).

La fase STT è risultata sensibile soprattutto alla durata dell'audio in ingresso, con intervalli osservati di 1–3 s in locale Windows e 1–4 s nel setup Ubuntu testato. Per la fase RAG/LLM non è disponibile una misura per-fase sistematica: in questa versione manca una telemetria persistente per turno che separi in modo robusto retrieval e generazione.

Nel campione osservato, il collo di bottiglia percepito è la fase TTS in termini di tempo al primo audio: circa 5 s in locale e circa 7–8 s su server. La latenza end-to-end completa non è stata misurata sistematicamente; di conseguenza, in questo capitolo viene trattata come n.d. sul piano quantitativo.

5.2.2 Latenza end-to-end e stabilità dei servizi

Cold-start e turni successivi mostrano andamenti diversi: il primo indica il tempo necessario a rendere lo stack pronto, il secondo descrive il comportamento steady-state durante la conversazione. Le criticità e le mitigazioni emerse restano allineate a quanto descritto in sottosezione 4.4.1 e sottosezione 4.3.3.

I valori in Tabella 5.2 non coincidono con la latenza di un singolo turno: descrivono solo la fase iniziale di warmup/setup. Nei turni successivi, il fattore

Voce	Locale (Windows)	Server (Ubuntu)
Warmup/setup iniziale (stack pronto)	circa 30 s	circa 80 s

Tab. 5.2: Tempo di inizializzazione (cold-start) osservato per rendere i servizi pronti all’interazione.

più visibile per l’utente resta il tempo al primo audio TTS (circa 5 s in locale e circa 7–8 s su server), mentre la latenza end-to-end non è stata misurata sistematicamente.

Per la validità dei risultati, la stima sistematica della latenza end-to-end e della fase RAG/LLM richiede una telemetria turn-based persistente (ID turno e timestamp sincronizzati tra client e servizi). Nello stato attuale, i log disponibili consentono un confronto osservativo tra ambienti ma non una decomposizione quantitativa completa per ogni turno.

Per la stabilità, le modalità di guasto (failure mode) più ricorrenti nel campione osservato sono timeout client-side su turni complessi e degradazione prestazionale in condizioni di risorse limitate. Nelle sessioni esaminate non sono emersi blocchi strutturali della macchina a stati del client. Nel deploy server, però, i tempi oscillano di più quando si sommano overhead di rete/proxy e carico inferenziale.

5.2.3 Osservazioni tra ambiente locale e server

Il confronto locale/server mostra che il contratto applicativo resta invariato. Cambia invece il comportamento operativo: in locale il networking è più prevedibile, mentre su server emergono più spesso jitter, dipendenze infrastrutturali e variabilità delle risorse. Questo quadro conferma l’impostazione del Capitolo 3 e quanto discusso in sottosezione 4.3.2, sottosezione 4.4.5.

In ambiente locale i tempi riflettono quasi solo il costo di inferenza dei moduli STT, RAG/LLM e TTS. In ambiente server si aggiungono overhead di instradamento ed effetti di contesa sulle risorse, con impatto più evidente sulla coda dei tempi (turni lenti) rispetto ai casi medi.

La stessa build Unity mantiene coerenza funzionale nei due contesti grazie alla normalizzazione degli endpoint. Nel deploy pubblico serve però più attenzione alla robustezza operativa (health-check, readiness reale dei servizi e controllo dei colli

di bottiglia inferenziali). Questa distinzione aiuta a leggere i risultati: locale come riferimento tecnico di base, server come riferimento realistico d'uso.

5.3 Risultati qualitativi e casi d'uso

5.3.1 Qualità percepita dell'interazione

La qualità percepita dell'interazione vocale in SOULFRAME dipende dall'equilibrio tra naturalezza, reattività e continuità del turno. La letteratura sulla *Voice User Experience* (Voice UX) mostra che queste dimensioni vengono valutate con strumenti eterogenei e non sempre comparabili; per questo il capitolo adotta un taglio osservativo. Il testo descrive il comportamento del prototipo in condizioni d'uso realistiche e discute quali scelte progettuali riducono le frizioni della conversazione vocale.[23] I dettagli implementativi sono nel Capitolo 4 (sottosezione 4.1.8, sottosezione 4.2.3, sottosezione 4.1.7).

Per la voce sintetizzata, l'impiego di Coqui XTTS v2 con *voice cloning* rafforza subito la percezione di identità dell'avatar. Il servizio applica un preprocessing del testo (funzione `_clean_tts_text`) per rimuovere simboli e punteggiatura che rischierebbero di essere vocalizzati in modo innaturale, mantenendo i segni utili a guidare pause e intonazione. La sintesi usa parametri di campionamento e penalità di ripetizione (ad esempio `temperature`, `top_k`, `top_p`, `repetition_penalty`) tarati per privilegiare stabilità e scorrevolezza rispetto a variazioni eccessive, senza introdurre metriche numeriche di tipo MOS che richiederebbero valutazioni controllate. In questa sezione l'osservazione resta qualitativa; la discussione completa sui limiti di somiglianza percepita è riportata in sezione 6.3.

Quando la memoria è attiva, la coerenza delle risposte si nota soprattutto nella capacità dell'avatar di “parlare come la persona”. Il servizio RAG distingue contenuti fattuali e tracce di stile, individua note di tipo `persona_style` e costruisce un riepilogo compatto di `persona_cues` da affiancare al contesto recuperato. Il prompt di sistema impone inoltre risposte in prima persona, come se l'avatar fosse l'interlocutore reale, evitando etichette meta e indicazioni sceniche. Una regola di *grounding* limita l'introduzione di dettagli non presenti nella memoria e incoraggia risposte del tipo “Non ricordo” quando mancano elementi sufficienti. Questo vincolo incide in modo chiaro sulla qualità percepita: anche senza questionario

formale, l’utente tende a considerare più credibile un agente che ammette l’incertezza rispetto a uno che riempie i vuoti con contenuti plausibili ma infondati. Ho mantenuto questa scelta perché, in un contesto embodied, l’errore semantico viene percepito come incoerenza di identità e non come semplice imprecisione testuale.

La fluidità del turno conversazionale dipende anche da come il sistema gestisce i tempi morti tra fine input e inizio output vocale. In **MainMode** il client aggiorna lo stato con messaggi come “Sto pensando...” durante la richiesta a `/chat` e “Sto parlando...” prima dell’avvio della sintesi. In parallelo, la risposta viene generata in streaming tramite `/tts_stream`, così da iniziare la riproduzione appena arrivano i primi chunk audio. Per ridurre il silenzio percepito nella finestra iniziale, il client riproduce una *wait phrase* breve, selezionata da un insieme predefinito di chiavi (ad esempio “hm”, “beh”, “aspetta”, “si”, “un_secondo”) e scaricata on-demand dall’endpoint `/wait_phrase`, con generazione automatica se il contenuto non è ancora disponibile. Anche un micro-segnale immediato può bastare a far leggere l’attesa come elaborazione in corso e non come blocco. La stessa logica supporta l’interruzione: quando l’utente riprende la parola, il client può abortire richieste e playback in corso, evitando sovrapposizioni tra audio tardivo e nuovo turno. La diagnostica resta disponibile tramite `DebugText` nella **Status Bar** (disattivabile con tasto **INS** su desktop o con triplo tap della barra in modalità touch), così da separare feedback operativo per l’utente e dettagli tecnici per debug.

Il sottosistema visivo lavora soprattutto su atmosfera e continuità percettiva, non come canale diagnostico principale. Lo stato del turno viene comunicato prima da testo di stato, transcript/reply e **UIHintBar**; rings e testo azzurro lampeggiante (desktop) rafforzano leggibilità e identità visiva senza sostituire i segnali funzionali. Questi elementi sostengono coinvolgimento e riconoscibilità dell’agente, mentre la parte strettamente operativa resta affidata ai feedback testuali.[24]

Figura 5.1 documenta un esempio di interazione in **MainMode** su desktop: la schermata mostra insieme trascrizione dell’utente, risposta dell’avatar, testo di stato e grammatica dei comandi nella hint bar. L’immagine permette di discutere leggibilità dello stato e coerenza del mood senza ripetere dettagli implementativi

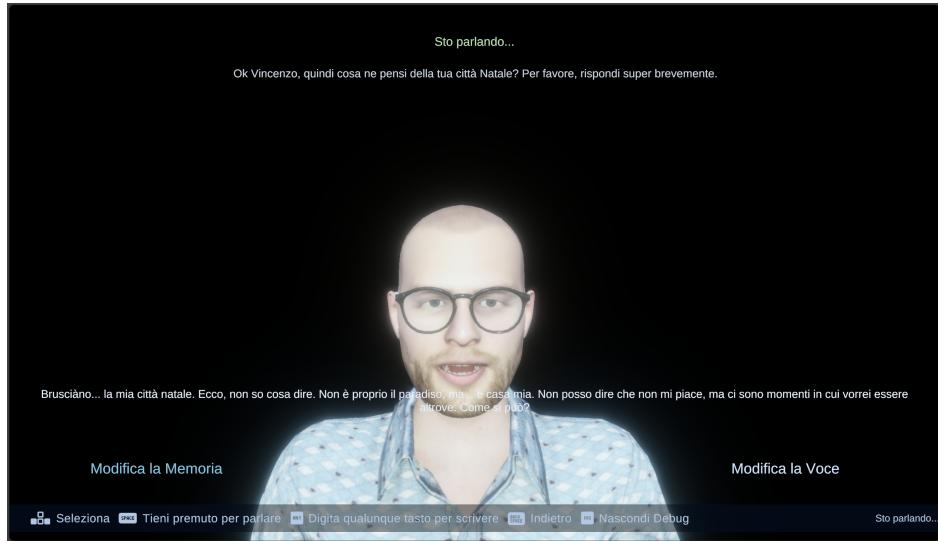


Fig. 5.1: Interfaccia desktop in modalità MainMode con transcript, risposta dell'avatar e stato conversazionale in forma testuale.

già trattati nei capitoli precedenti.

5.3.2 Usabilità interfaccia desktop e touch

L’usabilità dell’interfaccia in SOULFRAME è stata osservata confrontando due varianti che condividono lo stesso flusso logico, ma propongono affordanze diverse in base al dispositivo. In entrambe le modalità, il comportamento dell’applicazione è governato dagli stessi stati (`Boot`, `MainMenu`, `AvatarLibrary`, `SetupVoice`, `SetupMemory`, `MainMode`) e attiva la stessa pipeline backend per Speech-to-Text, Retrieval-Augmented Generation e Text-to-Speech. La differenza percepita nasce dalla superficie di interazione: desktop privilegia tastiera e mouse con suggerimenti contestuali tramite hint bar, mentre touch introduce controlli diretti (Push-to-Talk) e gesti (swipe) per ridurre l’ambiguità operativa su schermo. In implementazione, questa scelta passa da una rilevazione esplicita del profilo d’uso (`ShouldEnableTouchUi()`) e da un sistema di override che sostituisce pannelli e riferimenti a testi e bottoni senza modificare la macchina a stati. Così si mantiene coerenza funzionale e si riducono regressioni; i dettagli tecnici sono in sottosezione 4.1.1 e sottosezione 4.1.8.

Nella variante desktop l’azione principale è point-and-click, con supporto costante della hint bar per comandi e shortcut. Questa impostazione si presta a un

uso prolungato perché sfrutta abitudini già consolidate: l'utente individua rapidamente il focus (avatar e area testuale), mantiene controllo fine con il mouse e accelera azioni ricorrenti con la tastiera, soprattutto nei passaggi di setup. La gestione dell'attenzione basata su inattività del mouse (`mainModeMouseIdleSeconds`) riduce rumore visivo quando l'utente non interagisce, preservando la leggibilità dei contenuti principali. Per l'utente questo modello riduce il carico cognitivo: esplicita le azioni disponibili con suggerimenti persistenti e separa chiaramente canale di input (controlli e hint) e canale di output (risposta e feedback di stato).



Fig. 5.2: Interfaccia touch in modalità MainMode: Push-to-Talk, swipe tra transcript e reply e adattamenti per l'usabilità su schermo touch.

La variante

touch riorienta invece l'interazione attorno a un gesto centrale facile da memorizzare: il Push-to-Talk in **MainMode**. Il bottone dedicato (`btnTouchPttMainMode`) espone direttamente lo stato idle/active e riduce la necessità di ricordare scorciatoie o interpretare indicazioni testuali, condizione importante su schermi piccoli e in contesti d'uso intermittenti.

La separazione tra transcript e reply è gestita tramite swipe, con soglia minima (`touchMainModeSwipeMinDistance=70`).

La transizione

rapida (`touchMainModeTextSwitchDuration=0.18s`) rende

il cambio di vista percepibile ma non invasivo. Questo riduce il carico cognitivo: evita di comprimere troppo testo e controlli nello stesso viewport e lascia all'utente la scelta di portare in primo piano l'informazione più utile in quel momento (ciò che ha detto o ciò che ha ricevuto come risposta). Quando è presente input testuale, il dimming dell'avatar (`touchChatAvatarDimMultiplier=0.55`) riduce la competizione tra volto e area

di scrittura, spostando l'attenzione verso il contenuto operativo senza eliminare del tutto la percezione di presenza.

Le scelte di posizionamento di avatar e aree testuali costruiscono una gerarchia visiva pensata per mantenere stabile il focus. In entrambe le varianti, il volto dell'avatar resta un anchor percettivo: dà continuità tra i turni e sostiene l'impressione di interlocutore, mentre l'area della risposta deve restare leggibile e subito associabile alla produzione vocale. Su desktop, lo spazio disponibile permette di mostrare insieme transcript, reply e hint operativi senza perdere leggibilità. Su touch, la stessa simultaneità rischia di saturare lo schermo e aumentare lo scanning visivo; per questo swipe e layout adattato risultano più efficaci. Anche l'adattamento del CanvasScaler (`ConfigureTouchCanvasScalerForCurrentProfile()`) agisce su reference resolution e scale factor per preservare leggibilità su densità di pixel diverse.

Desktop e touch mostrano trade-off netti: desktop favorisce rapidità e trasparenza grazie a spazio informativo e hint persistenti; touch privilegia immediatezza del gesto, ma con minore densità simultanea di informazioni. La Figura 5.2 rende visibile questo equilibrio: tap per Push-to-Talk e swipe semplificano l'azione principale, mentre il dimming dell'avatar durante l'input testuale riduce la competizione visiva.

5.3.3 Analisi di casi e failure cases

I meccanismi tecnici richiamati nei tre casi sono analizzati in dettaglio nel Capitolo 4, in particolare sottosezione 4.2.2, sottosezione 4.2.3 e sottosezione 4.3.3. Durante la valutazione esplorativa sono emersi casi rappresentativi utili a capire dove la pipeline resta robusta e dove, invece, la combinazione tra rete, risorse di inferenza e qualità del contesto recuperato produce degradazioni percepibili. Sono stati selezionati tre casi (positivo, intermedio, critico) perché coprono funzionamento nominale, degradazione progressiva e failure esplicito della pipeline. I casi descritti di seguito non riportano trascrizioni o dialoghi letterali, ma sintetizzano input, comportamento e cause tecniche osservate, collegandoli ai meccanismi di fallback e ai vincoli di timeout e retry presenti nel sistema.

Il primo caso, positivo, riguarda una domanda che richiede l'uso della memoria per-avatar e in cui il contesto recuperato è pertinente e non ambiguo. Lo scenario

tipico è una richiesta su un'informazione già ingestita (ad esempio note biografiche, preferenze o episodi), formulata in modo abbastanza specifico da attivare retrieval mirato. In questa condizione `rag_server.py` esegue la ricerca ibrida e costruisce un prompt con un numero limitato di chunk (`RAG_PERSONA_TOP_K`) che, dopo deduplicazione, risultano coerenti tra loro. L'output generato dall'LLM appare allineato sia ai fatti riportati nella memoria sia allo stile dell'avatar: la componente di persona-style viene riconosciuta tramite detection dedicata e tradotta in cue che orientano tono e scelte lessicali. L'effetto percepito è una risposta che suona naturale e viene attribuita all'identità dell'avatar, senza oscillazioni meta o riferimenti a ruoli di sistema. Contribuisce anche la sanitizzazione finale della risposta (`_sanitize_chat_answer`), che rimuove prefissi di parlante e stage directions evitando che il TTS vocalizzi annotazioni non destinate all'utente. La pipeline visiva aiuta la leggibilità del turno: rings e post-processing segnalano chiaramente la transizione listening→processing→speaking, mentre lo streaming TTS riduce l'intervallo di silenzio tra fine utterance e inizio playback.

Il secondo caso, intermedio, emerge quando l'informazione richiesta è pertinente alla memoria ma il retrieval produce un contesto rumoroso o parzialmente ridondante. La causa più frequente è una domanda formulata in modo ampio, che attiva chunk con similarità comparabile e contenuti solo debolmente correlati tra loro. In queste condizioni la deduplicazione basata su soglia (`similarity>0.92`) riduce parte della ridondanza, ma può lasciare frammenti semanticamente sovrapposti o, al contrario, eliminare pezzi utili quando la similarità geometrica non riflette l'importanza informativa. Il comportamento osservato è una risposta generalmente corretta nel tema, ma con dettagli incompleti, generalizzazioni o piccole incoerenze di stile tra frasi successive, soprattutto quando nel contesto coesistono note fattuali e note di persona-style non perfettamente separate. Questo caso mostra che la qualità del risultato dipende più dalla qualità del set di chunk selezionati che dal solo vincolo di grounded mode: anche rispettando il principio di non introdurre affermazioni esterne, un contesto impreciso può guidare verso una risposta che appare “sfocata”. In questi casi è rilevante la possibilità operativa di ripulire la memoria per-avatar tramite `/clear_avatar`, quando l'ingestione precedente ha introdotto frammenti incoerenti o duplicati che degradano sistematicamente il retrieval. Dal lato UX, i feedback visivi continuano a segna-

lare correttamente lo stato della pipeline, ma non comunicano la qualità interna del contesto; l’utente percepisce quindi l’esito come risposta “meno centrata” pur in presenza di un ciclo conversazionale fluido.

Il terzo caso, critico, riguarda modalità di guasto legate a timeout e recupero incompleto, con impatto diretto sulla continuità dell’interazione. Lo scenario più semplice è un fallimento in STT: `whisper_server.py` non implementa retry interno e, se `/transcribe` non risponde in tempo o fallisce per errori temporanei, l’onere ricade sul client Unity. Con `requestTimeoutSeconds=15` e `retryCount=1`, un singolo ritentativo può risultare insufficiente in condizioni di carico elevato o audio più lungo del previsto; il comportamento osservato è quindi una transizione verso stato di errore o un ritorno in listening senza trascrizione valida. In modo analogo, la fase TTS può degradare quando si verifica un errore GPU: `coqui_tts_server.py` tenta un fallback CUDA→CPU (`_switch_to_cpu_fallback`), ma se l’errore avviene durante lo streaming o se la generazione non produce chunk validi, il server può emettere un silenzio di sicurezza. Questa scelta evita connessioni aperte senza output, ma genera un’esperienza ambigua se non è accompagnata da un messaggio UI chiaro: l’utente vede rings e stato speaking, ma non sente audio, oppure sente un frammento seguito da interruzione. Il client prova a mitigare l’attesa con wait phrases; in un failure prolungato, però, l’effetto si inverte e il riempitivo può diventare ripetitivo, segnalando che il turno non si chiude. In questi casi il comportamento dei feedback visivi diventa cruciale: una variazione coerente dei rings e dell’intensità del bloom può segnalare che il sistema sta tentando un recupero (retry o fallback), ma se non differenzia abbastanza processing normale e processing in errore, l’utente non distingue tra lentezza e guasto.

Figura 5.3 propone un diagramma di sequenza che sintetizza un failure case centrato su timeout in STT, ritentativo lato client ed eventuale uscita verso messaggio di errore o ritorno a listening. Il diagramma rende esplicite le dipendenze tra timeout, retry e feedback percettivo (stato UI e rings), chiarendo dove un recupero tecnico può non tradursi in un recupero dell’esperienza.

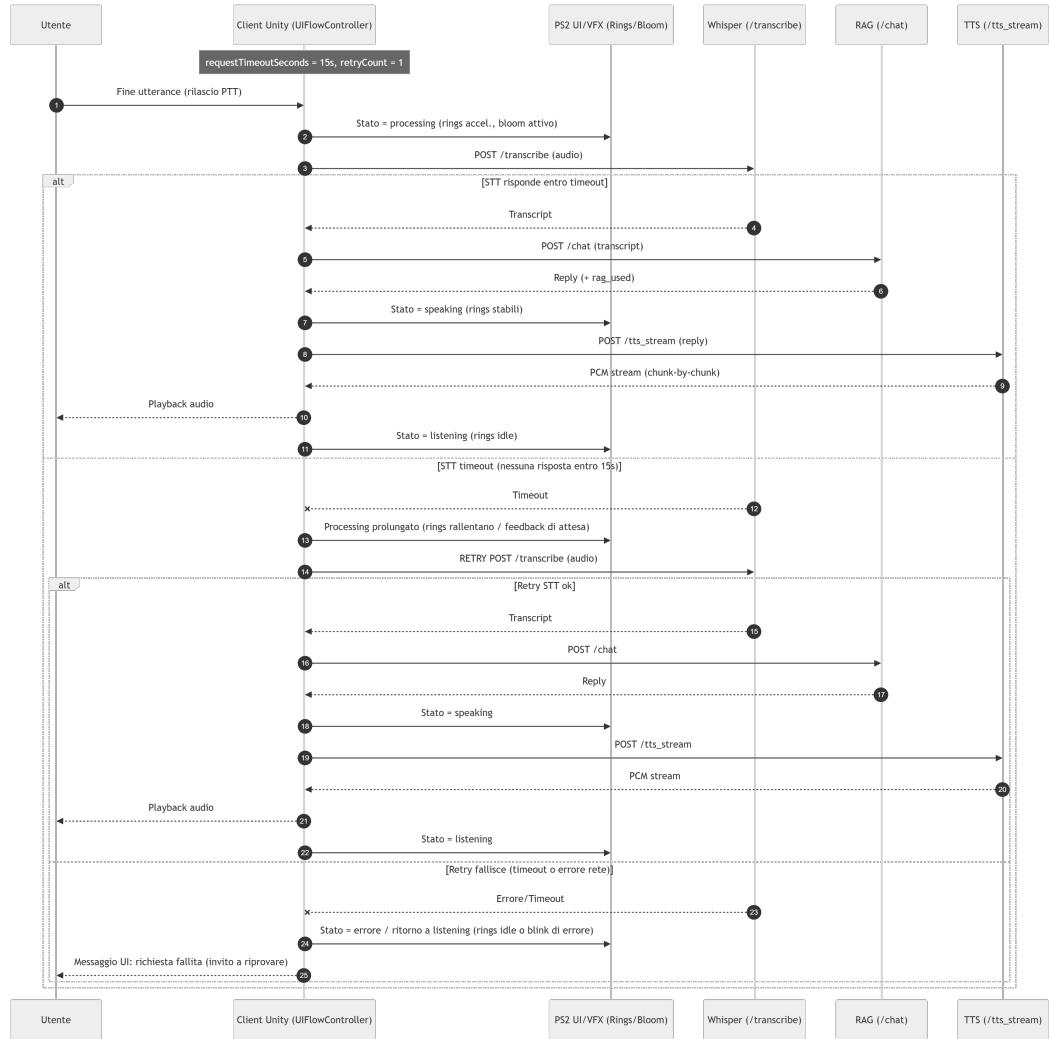


Fig. 5.3: Diagramma di sequenza di un failure case con timeout, retry e feedback visivo.

5.4 Discussione dei risultati

5.4.1 Punti di forza

La modularità resta un punto di forza: i servizi Speech-to-Text (STT), Retrieval-Augmented Generation (RAG) e Text-to-Speech (TTS) sono separati, raggiungibili tramite endpoint dedicati e verificabili con controlli di stato (`/health`). Questa scomposizione semplifica diagnosis e manutenzione perché mantiene localizzabili i colli di bottiglia per componente. La stessa struttura rende coerente il passaggio tra ambiente locale e server: l'orchestrazione del

client non cambia, cambia la configurazione degli endpoint.

Sul piano dell’esperienza, lo streaming TTS anticipa l’avvio dell’audio ai primi chunk PCM, senza attendere la sintesi completa. Nei turni osservati, con tempo al primo audio intorno a 5 s in locale e 7–8 s su server, questo passaggio rende l’attesa più leggibile. Le wait phrases non accorciano il tempo di inferenza, ma trasformano il silenzio in un segnale conversazionale interpretabile come presa di turno.

La qualità del recupero beneficia della ricerca ibrida (segnali lessicali BM25 + similarità vettoriale), che migliora la pertinenza in presenza di nomi propri, parole chiave e parafrasi. L’isolamento della memoria per-avatar e la deduplicazione dei chunk limitano interferenze e ridondanze. Sul piano operativo, i fallback mantengono il ciclo utilizzabile anche in condizioni non ideali: degradazione TTS su CPU, gestione dei chunk falliti, recovery lato retrieval e timeout/retry lato client.

Anche l’interfaccia contribuisce alla robustezza complessiva: desktop e touch condividono la stessa macchina a stati, ma usano affordance diverse in base al dispositivo. Questo riduce divergenze implementative e mantiene stabile il modello mentale dell’utente. Il contributo dei feedback visivi resta coerente con questa logica: sostiene leggibilità dello stato e identità dell’esperienza, senza sostituire i canali informativi primari.

5.4.2 Criticità osservate

Durante le sessioni esplorative sono emerse tre criticità ricorrenti che non rientrano nei failure case tecnici descritti in precedenza, ma che incidono sulla qualità percepita dell’interazione.

In alcuni turni l’avatar ha prodotto affermazioni non presenti nella memoria indicizzata, soprattutto con query vaghe e contesto recuperato debole. Ho osservato quindi risposte plausibili ma non allineate a quanto realmente memorizzato. L’inquadramento analitico del fenomeno è in sezione 6.3.

Con PDF a struttura articolata (tabelle, layout multi-colonna), il retrieval è apparso meno affidabile rispetto a note lineari o testo semplice. Il sintomo ricorrente è la restituzione di frammenti parziali o poco ordinati anche con documento

indicizzato. I dettagli tecnici sono in sottosezione 4.4.3, mentre la discussione valutativa completa è in sezione 6.3.

La voce sintetizzata è risultata comprensibile e riconoscibile come voce dell'avatar, ma con somiglianza variabile rispetto al campione originale. L'effetto non blocca l'interazione, però riduce la coerenza d'identità percepita nei turni più lunghi. La trattazione completa resta in sezione 6.3.

5.4.3 *Limiti emersi*

I limiti emersi sono coerenti con le criticità di implementazione già tracciate nel Capitolo 4 (sottosezione 4.4.1, sottosezione 4.4.3, sottosezione 4.4.4). Il limite più evidente nei turni osservati è il tempo al primo audio TTS: anche con streaming e strategie di mascheramento, il valore resta nell'ordine di circa 5 s in locale e circa 7–8 s su server. La latenza end-to-end completa non è stata misurata sistematicamente e resta quindi n.d. sul piano quantitativo. La fase RAG/LLM resta il principale fattore non isolato: la dipendenza dal runtime locale di inferenza (Ollama) e dal profilo del modello quantizzato rende la prestazione sensibile a disponibilità di risorse e contesa. Nel deploy su server si aggiungono overhead di rete e proxy, che introducono jitter e allungano la coda dei tempi, aumentando i casi limite in cui un turno satura il timeout client-side o richiede ripetizione.

La valutazione dell'esperienza resta esplorativa sul piano metodologico: le conclusioni su naturalezza, co-presenza e impatto dei feedback visivi derivano da osservazioni qualitative. Questo approccio guida le iterazioni progettuali, ma non consente inferenze robuste su efficacia o gradimento, anche perché la Voice UX è spesso misurata con strumenti eterogenei e non sempre validati in modo uniforme.[23]

Sul versante delle metriche tecniche, la Word Error Rate (WER) non è stata misurata in modo sistematico su un corpus di riferimento. In locale Windows lo STT usa Whisper `small`; nel setup server Ubuntu testato usa `medium`, reso praticabile da una VRAM maggiore (circa 24 GB contro circa 12 GB in locale). L'assenza di una valutazione quantitativa impedisce comunque di stimare con precisione la sensibilità a rumore, accenti e condizioni acustiche avverse, e di confrontare in modo controllato taglie di modello diverse.[7] Anche il modulo RAG mostra limiti strutturali: la qualità della risposta dipende dalla qualità

dei chunk e dal rumore introdotto da OCR o ingestione non curata; inoltre, nonostante regole di grounded mode, possono persistere errori di faithfulness o generalizzazioni quando il contesto recuperato è incompleto o ambiguo.[22] In modo analogo, l'OCR su documenti complessi (tabelle dense, layout non lineare, scansioni degradate) resta fragile e può introdurre testo rumoroso che si propaga al retrieval.

Infine, persistono limiti di portabilità pratica legati al runtime: su Windows la persistenza di ChromaDB può incorrere in lock e handle concorrenti, mitigati con procedure dedicate ma non eliminati del tutto. In generale, la riproducibilità dipende ancora dalla corretta installazione di dipendenze Python e dalla configurazione di componenti opzionali come OCR e servizi esterni; questi aspetti richiedono hardening ulteriore per ridurre la variabilità tra macchine.

5.4.4 Sintesi rispetto alle research questions

Le tre research questions, definite nel Capitolo 1, leggono i risultati su tre assi: fattibilità tecnica, qualità percepita e riproducibilità. Per RQ1, le evidenze indicano che il prototipo realizza un ciclo end-to-end stabile (STT → RAG/LLM → TTS → output audiovisivo) sia in locale sia in deploy, includendo memoria per-avtar e ingestione documentale con OCR. Nella configurazione osservata questo ciclo usa Whisper `small` in locale Windows e `medium` nel setup Ubuntu testato, coerentemente con la diversa VRAM disponibile. La risposta resta parziale sul lato quantitativo: servono campagne più ampie su tempi e accuratezza. Le basi progettuali di RQ1 sono in sezione 3.2, mentre i dettagli esecutivi sono nel Capitolo 4.

Per RQ2, i dati qualitativi suggeriscono che voice cloning, gestione del turn-taking (streaming + wait phrases) e segnali visivi di stato migliorano la fluidità percepita del dialogo. Queste evidenze vanno lette come indicatori qualitativi su co-presenza e naturalezza, utili per una valutazione successiva con protocollo dedicato. Il riferimento teorico è nel Capitolo 2, mentre l'evidenza osservativa è raccolta in sezione 5.3.

Per RQ3, i risultati sono più solidi: la scomposizione in micro-servizi, i contratti HTTP esplicativi, i health-check e la configurazione centralizzata degli endpoint supportano modularità e replicabilità tra ambienti. Restano però limiti prati-

ci (dipendenze Python, lock su Windows, variabilità hardware/inferenziale) che impediscono una riproducibilità pienamente “push-button” e richiedono ulteriore hardening. L’impostazione nasce dai requisiti di manutenibilità/portabilità in sezione 3.1 ed è operazionalizzata nel Capitolo 4.

6. CONCLUSIONI E SVILUPPI FUTURI

6.1 *Sintesi del lavoro svolto*

SOULFRAME nasce da un problema concreto: far funzionare una conversazione vocale credibile dentro un ambiente 3D WebGL, senza ridurre l’avatar a un semplice “parlante” scollegato dalla scena. Nel prototipo, l’avatar è trattato come un profilo operativo completo, cioè aspetto, voce e memoria.

L’architettura è stata costruita su una pipeline STT–RAG/LLM–TTS e su una separazione netta tra frontend Unity WebGL e backend a micro-servizi. Questa impostazione mi ha permesso di isolare le responsabilità (trascrizione, orchestrazione conversazionale, sintesi vocale, gestione asset) e di mantenere lo stesso flusso applicativo in locale Windows e su server Ubuntu.

Un punto importante emerso durante lo sviluppo è che molte scelte non sono state “a tavolino”. La selezione finale dei modelli (Whisper, **11ama3:8b** quantizzato Q4, Coqui XTTS v2, **nomic-embed-text**) è arrivata dopo tentativi reali, con cambi di rotta anche costosi in termini di tempo.

Le misure quantitative dbloccoisponibili e i relativi limiti metodologici sono raccolti nel Capitolo 5. Le evidenze discusse derivano da valutazioni tecniche svolte tra macchina locale e server Google Cloud. Qui l’obiettivo è chiudere il percorso in modo più onesto: cosa ha funzionato davvero, cosa è stato ridimensionato e quali sviluppi hanno senso nel perimetro del progetto.

6.2 *Contributi principali*

Il contributo più solido è architettonico: una pipeline end-to-end leggibile, con contratti HTTP esplicativi, `/health` su ogni servizio e normalizzazione degli endpoint lato client. Questo ha ridotto l’accoppiamento tra componenti e ha reso

più semplice capire dove nasce un errore quando la stessa build passa da loopback locale a reverse proxy.

Un secondo contributo riguarda il processo di selezione dei modelli, che è stato tutt'altro che lineare. In una fase iniziale avevo impostato il TTS su Chatterbox (ResembleAI), ma dopo giorni di conflitti tra pacchetti e problemi di compatibilità con CUDA 12.8, aggravati dalla GPU RTX 5070 Ti molto recente, ho preferito interrompere quel percorso e passare a Coqui XTTS v2. Non è stata una scelta “elegante” sulla carta, ma è stata quella che ha sbloccato davvero il progetto.

Sul frontend, la scelta di governare desktop e touch con una FSM unica, affiancata da `UINavigator` e `UIHintBar`, ha mantenuto coerente il comportamento dell’interfaccia anche con input diversi. Nella conversazione, streaming TTS e wait phrases non eliminano i tempi di inferenza, ma evitano lunghi vuoti percepiti e rendono più leggibile lo stato del turno.

Infine, il contributo operativo: runbook e script separati per locale Windows e deploy Ubuntu, più gestione avatar pensata per i vincoli WebGL. Spostare import/list/serving degli avatar sul backend ha ridotto la fragilità lato browser, soprattutto quando entrano in gioco cache, persistenza e sincronizzazione del filesystem virtuale.

6.3 Limiti attuali del sistema

Il primo limite è metodologico. Come discusso nel Capitolo 5, alcune metriche restano n.d. perché manca una telemetria turn-based persistente e sincronizzata tra client e servizi. Me ne sono accorto presto: senza timestamp per turno, la diagnosi funziona per debugging operativo, ma non basta per una valutazione comparativa robusta.

Un secondo limite è nella memoria conversazionale. Oggi il sistema memorizza in modo affidabile quando c’è un trigger lessicale esplicito (per esempio “ricorda che...”), ma non estrae ancora in autonomia le informazioni davvero memorabili dal dialogo libero. Ho scelto di non forzare una versione “semi-intelligente” perché il rischio di falsi positivi in memoria, in questa fase, era più dannoso del beneficio.

Un terzo limite riguarda la generazione facciale ad-hoc. L’idea iniziale era usare PRNet (TensorFlow) per generare volti personalizzati, ma in pratica il flusso non ha raggiunto una qualità accettabile: Blender non gestiva bene la

separazione dei vertici del modello generato e il risultato richiedeva texture fittizie poco convincenti. A quel punto il rapporto costo/beneficio era sbilanciato, quindi ho preferito Avaturn per ottenere qualità visiva stabile con complessità molto più bassa.

Restano poi i vincoli del runtime WebGL: nel prototipo i test operativi si sono concentrati su Microsoft Edge (Chromium), mentre altri browser non sono stati caratterizzati in modo sistematico. Inoltre, la presenza di `fs.jslib` e dell'uso di `FS.syncfs` rende esplicito che la persistenza nel browser richiede passaggi aggiuntivi e può introdurre stati intermedi.

La resa del `lipsync` resta sensibile all'integrazione concreta tra Avaturn, `uLipSync` e condizioni WebGL. Anche con fallback e adattamenti sul mapping dei blendshape, nel prototipo non è stato possibile ottenere una sincronizzazione uniforme in tutte le combinazioni di avatar e carico.

Tra i limiti emersi resta la tendenza del modello linguistico a colmare vuoti di contesto con contenuto plausibile quando il retrieval non recupera materiale davvero utile. In sottosezione 5.4.2 questo comportamento è descritto dal punto di vista osservativo; qui lo inquadro come limite strutturale del bilanciamento tra fluidità conversazionale e controllo semantico. In un contesto embodied, anche una deviazione plausibile pesa più di un errore testuale isolato perché viene attribuita all'identità dell'interlocutore. La mitigazione richiede controlli aggiuntivi di faithfulness o politiche più conservative quando il contesto è sottosoglia, ma entrambe aumentano latenza o complessità nel perimetro attuale.

Quando i PDF ingestiti contengono tabelle articolate, layout multi-colonna o scansioni a bassa qualità, l'OCR produce testo frammentato e spesso riordinato in modo non lineare. Quei chunk rumorosi finiscono comunque nello store vettoriale e possono essere recuperati con similarità alta anche se il contenuto è parziale o invertito. In pratica il retrieval risponde, ma il contesto che costruisce può essere poco utilizzabile per la generazione, con risposte confuse o dichiarazioni di incertezza anche su documenti presenti. Per risolvere davvero questo punto serve un pre-processing layout-aware prima dell'indicizzazione, che nel prototipo non è stato ancora integrato.

Anche la qualità del voice cloning resta un compromesso. Coqui XTTS v2 produce una voce riconoscibile come sintetica e, su frasi lunghe o punteggiatura

complessa, timbro e prosodia possono allontanarsi dal campione registrato. La causa principale è la combinazione tra campione breve raccolto in app e vincoli computazionali di un setup consumer: estendere durata e qualità del reference migliorerebbe la resa, ma allungherebbe onboarding e inferenza oltre un equilibrio sostenibile sulla macchina di sviluppo (laptop AMD Ryzen AI 9 HX 370 con RTX 5070 Ti Mobile 12 GB VRAM). In questo contesto XTTS v2 resta la scelta più praticabile per voice cloning zero-shot locale, ma la somiglianza percepita va trattata come approssimazione e non come fedeltà.

Infine c'è un limite operativo che non va nascosto: la toolchain Python/CUDA resta fragile quando cambiano dipendenze o driver. A questo si aggiunge un hardening ancora parziale (autenticazione e rate limiting non completi), coerente con un prototipo single-user ma non ancora con un'esposizione estesa.

6.4 Sviluppi futuri prioritari

6.4.1 Miglioramenti tecnici del prototipo

Il primo sviluppo utile è una memoria conversazionale davvero intelligente. L'obiettivo è superare il trigger esplicito e introdurre un modulo che estragga automaticamente le informazioni rilevanti dal dialogo (ad esempio NER, intent detection o un secondo LLM leggero in pipeline), mantenendo però regole conservative per evitare memorizzazioni spurie.

Il secondo sviluppo riguarda la generazione avatar personalizzata da foto reale. Qui ha senso riaprire il filone PRNet con strumenti più maturi, oppure valutare alternative recenti (gaussian splatting o NeRF leggeri) che oggi sono più accessibili. La priorità non è “fare di più”, ma ottenere un salto percettivo reale senza riaprire la stessa complessità che aveva bloccato il primo tentativo.

Il terzo sviluppo è il porting mobile/Android. L'interfaccia è già impostata con logica touch-first in vari passaggi, ma l'inferenza resta legata a un backend dedicato. Un'architettura ibrida (client mobile + modelli lato cloud) può abbassare la barriera d'accesso senza comprimere eccessivamente la qualità.

Accanto a questi punti, resta prioritario consolidare la base tecnica: telemetria per turno, logging strutturato, readiness-check più rigorosi e hardening operativo

minimo del deploy. Sono interventi meno “visibili”, ma necessari per trasformare le osservazioni qualitative in evidenze misurabili.

6.4.2 Estensione della valutazione utenti

Il quarto sviluppo, probabilmente il più importante sul piano scientifico, è una valutazione con utenti reali su campione strutturato. Un protocollo controllato permetterebbe un confronto diretto con la letteratura su naturalezza e co-presenza e renderebbe più solidi i confronti tra iterazioni.

Il disegno sperimentale dovrebbe separare almeno tre dimensioni: usabilità dell’interfaccia, qualità percepita della conversazione e tolleranza ai ritardi. In questo modo si evita che un solo fattore dominante, come il tempo al primo audio, nasconde problemi diversi di navigazione, feedback o chiarezza dello stato.

Anche uno studio piccolo, se progettato bene, sarebbe già un passo avanti netto: renderebbe confrontabili le iterazioni successive e permetterebbe di decidere gli interventi futuri su basi meno impressionistiche.

6.5 Considerazioni finali

SOULFRAME mostra che una conversazione vocale embodied in WebGL è praticabile, ma solo se architettura, implementazione e operatività vengono pensate insieme. Il valore del lavoro non sta in una singola tecnica, ma nel fatto che il ciclo end-to-end resta osservabile, debuggabile e migliorabile.

Ho privilegiato scelte incrementali e verificabili: prima chiudere un flusso completo che funzionasse davvero, poi aumentare qualità e copertura delle misure. Se dovessi sintetizzare il percorso in una frase, direi questa: meno promesse “perfette”, più iterazioni concrete che reggono alla prova dell’uso reale.

RINGRAZIAMENTI

Scrivere questi ringraziamenti è strano: sulla carta è la parte più "leggera" della tesi, ma è anche quella in cui non posso nascondermi dietro la tecnica. Qui non devo dimostrare niente, devo solo essere onesto.

Ringrazio prima di tutto me stesso. Sono partito alla grande, poi mi sono perso per strada: per necessità, per lavoro, per la vita che a volte decide i tempi al posto tuo. Ci ho messo cinque anni e passa e, sì, ho pensato di mollare tutto. Non per pigrizia, ma perché a un certo punto mi sembrava di stare rincorrendo una versione di me rimasta indietro. Per molto tempo mi sono sentito in colpa per l'essere fuori corso, come se avessi "sbagliato" qualcosa. Oggi la leggo diversamente: ho fatto quello che potevo, con quello che avevo, e soprattutto ho trovato un modo per riprendermi. Questa tesi per me è un punto di partenza: non per diventare un altro, ma per continuare senza sentirmi bloccato.

Ringrazio la mia relatrice, la Prof.ssa Paola Barra. Anche se siamo riusciti a vederci spesso di sfuggita, per me è stato davvero bello incontrarla: è una di quelle persone che si percepiscono "belle" nel modo più semplice e più raro del termine. Le auguro, come le ho sempre detto, di stare sempre bene e tranquilla. Ringrazio anche Ilaria e Attilio: due presenze inaspettate, preziose, che mi hanno incitato e incoraggiato nei momenti giusti. Li stimo molto, e mi porto dietro la loro buona energia.

Ringrazio i miei nonni, perché in modi diversi sono stati casa, presenza e sostegno. Ringrazio mio padre, per esserci stato e per tutto quello che mi ha insegnato, anche quando io non ero bravo a riconoscerlo. Ringrazio anche tutto ciò che mi ha formato, nel bene e nel male: alcune cose mi hanno dato slancio, altre mi hanno costretto a crescere. Non è stato sempre comodo, ma è stato reale.

Ringrazio le persone che ho incontrato lungo il percorso, perché spesso sono state loro a darmi coraggio quando io ne avevo poco. Un grazie enorme a Marco, per avermi sopportato, spronato e rimesso in riga con pazienza (anche quando ero

intrattabile). Grazie ad Antonio per quei "complimenti" ogni tanto, chiaramente al contrario, ma comunque capaci di farmi sorridere quando serviva. Ringrazio anche Vincenzo per essere un bruscanese d'eccellenza, per essere uno dei volti biondi del progetto e per avermi accolto con una naturalezza che non dimentico: grazie davvero, e sappi che io farò lo stesso, sempre. Ringrazio Daniele e Federica per le pizze buone e, soprattutto, per avermi insegnato una tecnica sottile ma importantissima: essere un pelino più leggero, quando serve. Ringrazio Roberto e Roberta, i due roberti, per essere affascinanti e per quella presenza che resta, anche quando non si fa rumore.

Grazie ai colleghi, in particolare Vincenzo e Simone: sostenersi a vicenda non è mai abbastanza, eppure per un po' lo è stato. I chiarimenti anche a orari assurdi, le risate quando eravamo cotti, e quei pasti post-esame esageratamente calorici che per un periodo sono stati una specie di rito.

Ringrazio anche Livio e Francesca, il mio vecchio datore di lavoro e sua moglie: per quello che mi hanno dato, per quello che mi hanno insegnato e per l'aiuto nei momenti bui. E ringrazio tutte le persone che ci sono state, in qualsiasi forma: con un messaggio, una battuta, una spinta, una mano tesa. Anche quando magari non lo sapevano, hanno contatto.

Un ringraziamento, un po' storto ma sincero, va anche alla mia ansia. Mi ha bloccato tante volte, però in questi anni mi sono fatto forza davvero. Oggi capita sempre più spesso che sia lei ad avere paura di me. Non so come andrà il giorno della presentazione (magari la voce trema, magari no), ma so che non sono più la stessa persona che ha iniziato. E questo, da solo, vale tanto.

Infine ringrazio **SOULFRAME**. Non è stato solo un progetto: per me è rimasto un simbolo. È stata una sfida tecnica e personale, anche per la complessità e per tutto quello che mi ha costretto a reimparare da capo. L'ho vissuta con quella tipica dose di ossessione e attaccatura che mi prende quando ci tengo davvero, ma proprio per questo è una delle cose di cui vado più fiero.

Ci ho messo tempo, sì. Ma oggi posso dirlo: ne è valsa la pena.

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>.
 - [12] Avaturn. *Avaturn Unity WebView SDK – Example Scenes*. Accessed: 2025. 2024. URL: <https://github.com/avaturn/avaturn-unity-examples>.

-
- [13] Jiapeng Wang e Yihong Dong. «Measurement of Text Similarity: A Survey». In: *Information* 11.9 (2020). Open access (MDPI). Copre Levenshtein, Jaccard, e altre metriche string-based e corpus-based, p. 421. DOI: [10.3390/info11090421](https://doi.org/10.3390/info11090421). URL: <https://www.mdpi.com/2078-2489/11/9/421/pdf>.
 - [14] Nicola Dragoni et al. «Microservices: Yesterday, Today, and Tomorrow». In: *Present and Ulterior Software Engineering* (2017). Open access, arXiv:1606.04036, pp. 195–216. DOI: [10.1007/978-3-319-67425-4_12](https://doi.org/10.1007/978-3-319-67425-4_12). URL: <https://arxiv.org/pdf/1606.04036v4.pdf>.
 - [15] Stephen E. Robertson e Hugo Zaragoza. «The Probabilistic Relevance Framework: BM25 and Beyond». In: *Foundations and Trends in Information Retrieval* 3.4 (2009). Open access, pp. 333–389. URL: https://www.staff.city.ac.uk/~sbrp622/papers/foundations_bm25_review.pdf.
 - [16] Jianjun Chen et al. «We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS». In: *Proceedings of the 27th USENIX Security Symposium*. Open access via USENIX. USENIX Association, 2018, pp. 1079–1093. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/chen-jianjun>.
 - [17] Ray Smith, Daria Antonova e Dar-Shyang Lee. «Adapting the Tesseract Open Source OCR Engine for Multilingual OCR». In: *Proceedings of the International Workshop on Multilingual OCR (MOCR '09)*. Open access via Tesseract project. ACM, 2009. DOI: [10.1145/1577802.1577804](https://doi.org/10.1145/1577802.1577804). URL: <https://tesseract-ocr.github.io/docs/MOCRadaptingtesseract2.pdf>.
 - [18] Ying Wang et al. «Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem». In: *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. Open access via author page. ACM, 2020, pp. 125–135. DOI: [10.1145/3377811.3380426](https://doi.org/10.1145/3377811.3380426). URL: https://yepangliu.github.io/files/ICSE2020_Watchman.pdf.
 - [19] Bikash Barua. «A Microservices Approach to Fault Tolerance, Load Balancing and Service Discovery». In: *Preprint* (2024). Open Access. arXiv: [2410.19701](https://arxiv.org/abs/2410.19701). URL: <https://arxiv.org/abs/2410.19701>.

- [20] Rongxu Xu, Wenquan Jin e Dohyeun Kim. «Microservice Security Agent Based On API Gateway in Edge Computing». In: *Sensors* 19.22 (nov. 2019). Open Access, PMCID: PMC6891515, p. 4905. DOI: [10.3390/s19224905](https://doi.org/10.3390/s19224905). URL: <https://pmc.ncbi.nlm.nih.gov/articles/PMC6891515/>.
- [21] Md Shafiful Alam et al. «Securing RESTful APIs in Microservices Architectures: A Comprehensive Threat Model and Mitigation Framework». In: *International Journal of Engineering Research and Emerging Technology (IJERET)* 4.2 (2023). Open Access. DOI: [10.63282/3050-922X.IJERET-V4I2P107](https://doi.org/10.63282/3050-922X.IJERET-V4I2P107). URL: <https://ijeret.org/index.php/ijeret/article/view/124>.
- [22] Yunfan Gao et al. «Retrieval-Augmented Generation for Large Language Models: A Survey». In: *arXiv preprint arXiv:2312.10997* (2024). Full text disponibile su arXiv (open access). URL: <https://arxiv.org/abs/2312.10997>.
- [23] Katie Seaborn e Jacqueline Urakami. «Measuring Voice UX Quantitatively: A Rapid Review». In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems (CHI EA '21)*. Full text disponibile su arXiv (open access). ACM, 2021. DOI: [10.1145/3411763.3451712](https://doi.org/10.1145/3411763.3451712). URL: <https://arxiv.org/abs/2103.07108>.
- [24] Stacey Li et al. «The PUEVA Inventory: A Toolkit to Evaluate the Personality, Usability and Enjoyability of Voice Agents». In: *arXiv preprint arXiv:2112.10811* (2021). Full text disponibile su arXiv (open access). URL: <https://arxiv.org/abs/2112.10811>.