Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale Ingegneria del Cinema e dei Mezzi di Comunicazione

A.a. 2025/2026

Sessione di laurea Luglio 2026

Sintesi vocale espressiva per Virtual Humans

Relatore:

Prof. Andrea Bottino

Prof. Francesco Strada

Dott. Stefano Calzolari

Candidato:

Alessia Gianolla

Abstract

I virtual humans risultano tanto più efficaci quanto più l'utente li percepisce come credibili, e in questo la voce ha un ruolo centrale: oltre alla naturalezza e a una latenza contenuta, è anche l'espressività emotiva del parlato a sostenere il senso di presenza ed evitare che l'interlocutore ne avverta l'artificiosità. Nonostante i progressi della sintesi vocale (Text-to-Speech, TTS), permane però una forte carenza di soluzioni open source in lingua italiana integrabili nativamente in motori come Unity e dotate di controlli per la modulazione emotiva: una lacuna che costringe di fatto a dipendere da servizi cloud a pagamento.

Il presente lavoro propone una pipeline TTS che opera interamente in locale all'interno di Unity, eliminando la dipendenza da server esterni. Tra le soluzioni analizzate la scelta è ricaduta su Qwen3-TTS-0.6B, che ha mostrato capacità promettenti nella sintesi del parlato italiano. Trattandosi di un modello multistadio, una parte centrale del lavoro è consistita nell'esportarne i singoli componenti e adattarli all'esecuzione nativa in Unity Sentis. Viene inoltre proposta una metodologia per la valutazione soggettiva dell'espressività emotiva, fondata su confronti a coppie tra diversi modelli TTS e su tecniche di campionamento adattivo che riducono il numero di valutazioni necessarie a ordinarli in modo affidabile.

L'analisi delle prestazioni ha evidenziato una latenza contenuta sul primo segmento audio in modalità streaming, a fronte di un tempo di generazione complessivo ancora elevato, riconducibile principalmente alla fase di predizione multi-codebook, individuata come il collo di bottiglia del sistema. Sul piano percettivo, uno user study ha confrontato la pipeline con altri modelli open source basati su voice cloning; lo studio ha permesso di ordinare i modelli e individuare quello più efficace sul piano dell'inflessione emotiva. La pipeline è stata valutata nel contesto applicativo di InCare, un ambiente di formazione in realtà virtuale in cui il personale sanitario allena le proprie tecniche di comunicazione interagendo con un paziente virtuale.

Indice

Glossario IV

1 Introduzione 1

2 Background 3
2.1 Evoluzione della sintesi vocale neurale . . . 3
2.2 TTS end-to-end basati su Large Language Models . . . 7
2.3 Speech tokenization e codec neurali audio . . . 8
2.4 Streaming e latenza nei sistemi TTS . . . 11
2.5 Inferenza Machine Learning in Unity: ONNX e Sentis . . . 13
2.6 Sistemi TTS allo stato dell'arte . . . 15
2.6.1 Sistemi commerciali . . . 15
2.6.2 Open source . . . 16
2.6.3 Analisi comparativa e gap . . . 17

3 Architettura di Qwen3-TTS 19
3.1 Tokenizer . . . 21
3.2 LM Backbone (Talker) . . . 22
3.3 Modulo MTP . . . 25
3.4 Code2Wav . . . 25
3.5 Speaker Encoder . . . 27
3.6 Clonazione vocale . . . 27

4 Progettazione e implementazione della pipeline 29
4.1 Decomposizione della pipeline in componenti ONNX . . . 29
4.2 Metodologia di conversione Python-ONNX . . . 30
4.3 Esportazione dei componenti . . . 33
4.3.1 Talker . . . 33
4.3.2 Code Predictor . . . 36
4.3.3 Vocoder . . . 40
4.3.4 Speaker Encoder . . . 42

4.3.5 Embedding e text projection . . . 43
4.3.6 Sinossi delle incompatibilità e delle soluzioni . . . 43
4.4 Quantizzazione . . . 44
4.5 Validazione end-to-end: la pipeline Python di riferimento . . . 45
4.5.1 Inizializzazione e campionamento del primo token . . . 47
4.5.2 Il ciclo di generazione a doppio livello . . . 48
4.5.3 Sintesi in streaming . . . 50
4.6 Porting in C#/Unity/Sentis . . . 51
4.6.1 Tokenizzazione tramite libreria nativa . . . 52
4.6.2 Costruzione degli embedding fuori dall'ONNX . . . 53
4.6.3 Condizionamento vocale nella variante Base . . . 56
4.7 Analisi delle prestazioni . . . 56
4.7.1 Individuazione del collo di bottiglia . . . 57
4.7.2 Real-Time Factor e reattività . . . 58
4.7.3 Riduzione dei codebook residui . . . 59
4.7.4 Sintesi e implicazioni . . . 60

5 Valutazione soggettiva dell'espressività emotiva 62
5.1 Design sperimentale . . . 63
5.1.1 Stimoli . . . 63
5.1.2 Dataset di riferimento . . . 64
5.1.3 Compito del partecipante . . . 65
5.2 Il modello di ranking: TrueSkill . . . 66
5.3 Campionamento adattivo . . . 68
5.4 Implementazione e raccolta dei dati . . . 70
5.5 Risultati . . . 72
5.5.1 Partecipanti e dati raccolti . . . 72
5.5.2 Analisi delle stime di abilità . . . 73
5.6 Aggregazione dei punteggi: tre scenari a confronto . . . 75
5.7 Discussione e limiti . . . 77

6 Conclusioni 78

A Verifica della fusione degli embedding nel Talker 81

B Un esempio reale di campionamento adattivo 84

C Valori completi dei rating TrueSkill 87

Bibliografia 91

Glossario

TTS
: Text-to-Speech

LLM
: Large Language Model

VQ
: Vector Quantization

RVQ
: Residual Vector Quantization

GAN
: Generative Adversarial Network

GQA
: Grouped Query Attention

RoPE
: Rotary Position Embedding

MLP
: Multi-Layer Perceptron

MTP
: Multi-Token Prediction

ONNX
: Open Neural Network Exchange

RTF

Real-Time Factor

MOS

Mean Opinion Score

ICL

In-Context Learning

FFN

Feed-Forward Network

ASR

Automatic Speech Recognition

WER

Word Error Rate

M-RoPE

Multimodal Rotary Position Embedding

BPE

Byte Pair Encoding

KV

Key-Value

SPSS

Statistical Parametric Speech Synthesis

VITS

Variational Inference with adversarial learning for end-to-end Text-to-Speech

Capitolo 1

Introduzione

I *virtual humans* (agenti virtuali) trovano un impiego sempre più diffuso in ambiti come l'addestramento, la simulazione, l'assistenza e l'intrattenimento interattivo. In letteratura questi sistemi, dotati di un'incarnazione visiva e capaci di sostenere un dialogo multimodale con l'utente, vengono definiti *Embodied Conversational Agents* (ECA) [1]. La loro efficacia applicativa è legata in misura determinante alla *believability* (credibilità) [2], intesa come la capacità di rendere l'interazione talmente fluida e naturale da spingere l'utente a interagire con l'agente come se fosse un interlocutore reale, dimenticando temporaneamente la natura artificiale del sistema.

La creazione di un agente credibile va ben oltre la semplice qualità grafica del personaggio. Sebbene vi siano diversi fattori tecnici e di sviluppo che concorrono al realismo visivo, come l'accuratezza della modellazione 3D o la fluidità delle animazioni, la credibilità si fonda in realtà su un delicato equilibrio tra personalità, emotività, intenzionalità e intelligenza [3]. La carenza anche di un solo elemento è sufficiente a compromettere l'intera esperienza immersiva.

In questo equilibrio, la dimensione vocale ricopre una funzione decisiva, essendo il canale principale attraverso cui l'agente comunica le proprie emozioni. Per questo motivo, la tecnologia *Text-to-Speech (TTS)* assume un ruolo centrale: affinché il parlato sintetico sostenga la credibilità dell'agente, non è sufficiente che esso risulti naturale e intelligibile. Il sistema deve infatti operare con una latenza compatibile con i ritmi di una conversazione reale e veicolare un'espressività emotiva adeguata. Sebbene i recenti sviluppi nella sintesi vocale neurale abbiano reso il parlato artificiale quasi indistinguibile da quello umano per qualità acustica, il controllo dell'espressività rimane una sfida aperta [4], in particolar modo per le lingue diverse dall'inglese, data anche la scarsità di risorse dedicate [5].

Attualmente, il panorama tecnologico per la lingua italiana presenta una lacuna significativa. Le soluzioni in grado di offrire elevata qualità e un controllo emotivo sono prevalentemente servizi commerciali erogati in cloud, i quali impongono

la dipendenza da una connessione di rete, costi operativi ricorrenti e l'invio di dati a server esterni. Di contro, i sistemi open source eseguibili localmente sono storicamente concepiti per ambienti Python e hardware di classe workstation, risultando inadatti a un'integrazione diretta all'interno di game o rendering engine. Risulta pertanto complesso individuare una soluzione TTS per la lingua italiana che riesca a coniugare un elevato grado di espressività emotiva con la capacità di operare interamente *on-device* all'interno di un motore real-time.

Il presente lavoro si inserisce in questo contesto ed è motivato dalle necessità del progetto **InCare**, un ambiente di formazione in realtà virtuale destinato al personale sanitario. All'interno di questa piattaforma, l'operatore può affinare le proprie competenze comunicative interagendo con un paziente virtuale, che sostituisce i tradizionali attori impiegati nel training medico. In uno scenario simile, la credibilità emotiva del paziente è la condizione necessaria per l'efficacia formativa: l'operatore deve potersi confrontare con reazioni affettive verosimili, trasmesse in primo luogo attraverso la voce.

L'obiettivo primario di questa tesi è pertanto la progettazione e l'implementazione di una pipeline TTS operante interamente in locale all'interno del game engine Unity, eliminando qualsiasi dipendenza da servizi cloud. Tra i sistemi open source disponibili, la scelta è ricaduta su Qwen3-TTS (nella variante da 0,6 miliardi di parametri), per la qualità della sintesi in italiano e la predisposizione alla generazione a bassa latenza. Il raggiungimento di questo scopo ha richiesto l'esportazione dei modelli originali in formato Open Neural Network Exchange (ONNX), la risoluzione delle incompatibilità architetturali, l'adattamento dei tensori tramite la libreria Sentis e la ricostruzione in C# della logica di esecuzione.

A questo primo obiettivo ne segue un secondo di natura metodologica: la definizione di una procedura per la valutazione soggettiva, tramite user study, dell'espressività emotiva del parlato sintetico. La procedura proposta si basa su confronti a coppie tra diversi modelli TTS, collezionati e aggregati mediante un sistema di rating bayesiano [6] e guidati da una strategia di campionamento adattivo, al fine di ottenere un ranking affidabile minimizzando il numero di valutazioni necessarie.

Capitolo 2

Background

Il TTS ha l'obiettivo di generare, a partire da un testo, un segnale vocale quanto più possibile intelligibile e naturale. Per la sua versatilità trova impiego in numerose applicazioni di comunicazione uomo-macchina e ultimamente anche in contesti legati all'intelligenza artificiale e all'elaborazione del linguaggio naturale [7].

2.1 Evoluzione della sintesi vocale neurale

I primi sistemi di sintesi vocale [7] basati su calcolatore risalgono alla seconda metà del XX secolo. Si possono distinguere tre approcci principali: la sintesi articolatoria, la sintesi a formanti e la sintesi concatenativa.

La sintesi articolatoria riproduce il comportamento degli organi fonatori come labbra, lingua, glottide e tratto vocale per generare il parlato. La difficoltà di modellarne con precisione i movimenti e di raccogliere dati articolatori adeguati ne ha però limitato la qualità, rimasta inferiore a quella degli altri due approcci.

La sintesi a formanti genera il parlato applicando un insieme di regole che pilotano un modello sorgente-filtro semplificato. Le formanti sono i picchi di risonanza del tratto vocale, ossia le frequenze che vengono amplificate quando il suono delle corde vocali attraversa le cavità orali e nasali. In questo approccio una sorgente viene filtrata per riprodurre le formanti, agendo su parametri come la frequenza fondamentale e le componenti di rumore. Tali regole, formulate manualmente da esperti per imitare la struttura del parlato, sono però complesse da definire, e il risultato, seppur intelligibile, presenta numerosi artefatti e un timbro poco naturale.

La sintesi concatenativa si fonda, invece, su un ampio database di frammenti vocali pre-registrati: da unità sub-fonemiche fino a parole e frasi intere, selezionati e concatenati per comporre l'enunciato finale. Offre buona intelligibilità e un timbro autentico, ma richiede database molto estesi e fatica a riprodurre una prosodia fluida e delle inflessioni emotive naturali.

Per superare questi limiti è stata introdotta la sintesi parametrica statistica (Statistical Parametric Speech Synthesis (SPSS)), che non genera la forma d'onda concatenando segmenti, ma predice prima un insieme di parametri acustici e li converte poi in segnale. Una pipeline SPSS si articola in tre componenti: un modulo di analisi del testo, un modello acustico che predice i parametri acustici a partire dalle feature linguistiche e un vocoder che, a partire da tali parametri, ricostruisce la forma d'onda. Rispetto agli approcci precedenti, la SPSS guadagna in naturalezza, flessibilità e contenimento dei costi di raccolta dati, in quanto richiede un database più ridotto. La voce prodotta conserva tuttavia un carattere artificiale e un'evidente presenza di artefatti, restando lontana dalla qualità del parlato umano.

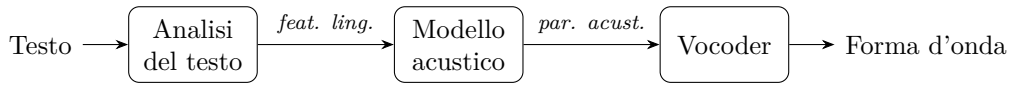


Figura 2.1: Schema a tre stadi di una pipeline SPSS

Con l'avanzamento del deep learning, si sono iniziate a sviluppare architetture TTS basate sulle reti neurali. Nel 2016 Google presenta WaveNet [8]: un vocoder neurale che, invece di ricostruire la forma d'onda a partire da parametri acustici con tecniche di signal processing, sfrutta una rete neurale per modellare direttamente i campioni audio, generandoli uno ad uno. La qualità supera per la prima volta quella dei sistemi concatenativi e parametrici, al prezzo però di una generazione lenta dovuta alla natura autoregressiva sul singolo campione, in cui ogni campione viene prodotto in sequenza, condizionato su tutti i precedenti, impedendo così la parallelizzazione del processo.

Nel 2017 Google introduce Tacotron [9], un modello generativo *end-to-end* che sintetizza il parlato direttamente dai caratteri di testo. A differenza delle architetture viste finora, un sistema end-to-end gestisce l'intera trasformazione dai dati di ingresso ai dati d'uscita tramite un'unica rete, eliminando la necessità di moduli separati addestrati in modo indipendente. L'innovazione principale di Tacotron consiste nel superare la pipeline multi-stadio tipica dei sistemi SPSS, sostituendola con un'architettura *sequence-to-sequence* in grado di trasformare direttamente una sequenza di caratteri di testo in una sequenza di frame di uno spettrogramma, ossia una rappresentazione tempo-frequenza del segnale audio. Questo approccio permette di addestrare il sistema da zero utilizzando semplici coppie di dati testo-audio. Poiché la sequenza di testo in ingresso e lo spettrogramma in uscita presentano lunghezze differenti e non possiedono un allineamento temporale a priori, l'architettura sfrutta un meccanismo di attenzione (*attention mechanism*). È proprio quest'ultimo a imparare dinamicamente su quali caratteri concentrarsi per generare ciascun frame acustico. Sebbene la qualità vocale risulti superiore a

quella dei sistemi parametrici statistici, il timbro generato da questa prima versione di Tacotron conserva ancora dei tratti artificiali. Ciò è dovuto principalmente al fatto che la forma d'onda finale non viene generata da un vocoder neurale, bensì ricostruita a partire dallo **spettrogramma lineare**[1] tramite l'algoritmo iterativo di Griffin-Lim [10].

L'anno successivo arriva Tacotron 2 [11], che unisce due reti neurali distinte per ottenere un parlato quasi indistinguibile da quello umano. La prima rete converte il testo in uno **spettrogramma Mel**[2] mentre la seconda, basata su WaveNet, ricostruisce da tale spettrogramma la forma d'onda finale. È proprio grazie alla sostituzione dell'algoritmo Griffin-Lim con il vocoder neurale che si riesce a colmare il divario di naturalezza lasciato aperto da Tacotron.

Nel 2019, Microsoft presenta FastSpeech [12], un modello acustico non-autoregressivo che genera l'intero spettrogramma Mel in parallelo anziché frame per frame. Al posto del meccanismo di attenzione dei modelli autoregressivi, fonte di instabilità ed errori come parole saltate o ripetute, adotta un predittore esplicito della durata dei fonemi, ottenendo allineamenti più robusti. Affronta così i due limiti di Tacotron 2, lentezza e fragilità, con una forte accelerazione della sintesi; resta però una pipeline a due stadi, dato che lo spettrogramma va ancora convertito in forma d'onda da un vocoder separato.

Sul fronte della sintesi della forma d'onda, l'introduzione del vocoder HiFi-GAN [13] nel 2020 rende la conversione da spettrogramma Mel a forma d'onda rapida ed efficiente grazie a un'architettura basata su reti generative avversarie (Generative Adversarial Network (GAN)). GAN è un modello generativo fondato sull'addestramento simultaneo e competitivo di due reti neurali distinte, un generatore e un discriminatore. Nel contesto della sintesi vocale, il generatore apprende a produrre forme d'onda sintetiche a partire dalle caratteristiche acustiche in ingresso, tentando di ingannare il discriminatore; quest'ultimo, di contro, viene addestrato a distinguere accuratamente i campioni audio generati da quelli reali appartenenti al dataset originale. Oltre a garantire un timbro estremamente naturale, HiFi-GAN abbatte drasticamente i tempi di generazione del vocoder grazie alla sua natura non-autoregressiva: a differenza dei sistemi sequenziali, come WaveNet, i campioni audio vengono generati interamente in parallelo.

---

[1] Spettrogramma in cui l'asse delle frequenze è espresso su scala lineare in Hz, ottenuto direttamente dalla trasformata di Fourier a breve termine (STFT). Conserva l'intera risoluzione spettrale del segnale, ed è la rappresentazione da cui l'algoritmo di Griffin-Lim stima iterativamente la fase per ricostruire la forma d'onda.

[2] Variante in cui le frequenze vengono mappate sulla scala Mel, una scala percettiva che approssima la risposta non lineare dell'orecchio umano. Rispetto allo spettrogramma lineare è più compatto e privilegia le regioni frequenziali più rilevanti per la percezione, al costo di una minore quantità di dettaglio.

Sebbene vocoder avanzati come HiFi-GAN avessero reso lo stadio di sintesi della forma d'onda rapido e di alta qualità, i sistemi TTS rimanevano ancorati a una pipeline a due stadi, con modello acustico e vocoder distinti. Per superare questo limite, nel 2021 viene introdotto Variational Inference with adversarial learning for end-to-end Text-to-Speech (VITS) [14], il primo metodo in grado di unificare modello acustico e vocoder all'interno di un'unica architettura end-to-end, addestrata congiuntamente. Nei modelli precedenti questi due stadi venivano tipicamente ottimizzati in modo separato. Generando la forma d'onda direttamente dal testo, VITS elimina la necessità di generare lo spettrogramma all'interno di un modulo per poi doverlo passare esplicitamente a un secondo: questo approccio previene l'accumulo di errori tra moduli e garantisce un parlato più naturale.

Un decisivo cambio di paradigma si verifica nel 2022, quando la generazione acustica inizia ad essere affrontata come un problema di *language modeling*: Google introduce AudioLM [15], un modello generativo che, dato un breve spezzone audio, ne produce una continuazione coerente. L'audio viene prima convertito in una sequenza di *token* discreti tramite codec neurale; il modello apprende quindi a prevedere i token successivi allo stesso modo in cui il modello linguistico prevede la parola seguente. I token predetti sono poi riconvertiti in forma d'onda dal codec. AudioLM dimostra per la prima volta la possibilità di generare parlato di alta qualità lavorando su token discreti anziché su feature continue come gli spettrogrammi.

Nel 2023 Microsoft estende questo paradigma al TTS con VALL-E [16]. Anche qui un codec neurale comprime la forma d'onda in token discreti, riducendo l'audio a una sequenza analoga a quella di un testo. Su questa rappresentazione il sistema applica le stesse architetture Transformer usate per il linguaggio scritto, imparando a generare i token audio a partire dal testo in ingresso. La voce da riprodurre è determinata da un breve *prompt* acustico: ne deriva la possibilità di clonare una voce mai sentita prima a partire da soli tre secondi di registrazione di riferimento (*zero-shot*).

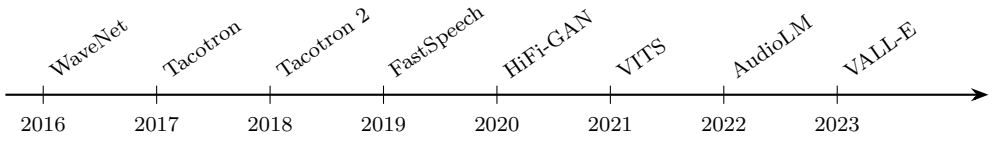


Figura 2.2: Principali tappe nell'evoluzione della sintesi vocale neurale.

2.2 TTS end-to-end basati su LLMs

Come visto nella sezione precedente, con AudioLM e VALL-E la generazione del parlato inizia a essere formulata come un problema di language modeling su token audio discreti. Questa sezione approfondisce tale paradigma, indicato come *LLM-TTS*, ripercorrendo i sistemi che lo hanno definito e consolidato, con particolare attenzione al modo in cui l'audio viene rappresentato in token, un aspetto che rappresenta la linea principale di evoluzione tra un sistema e il successivo. Il vantaggio di fondo di questa formulazione è poter riutilizzare per il parlato le tecniche e le architetture maturate sui modelli linguistici, con benefici in termini di naturalezza, capacità di clonare voci da un riferimento (zero-shot) e scalabilità rispetto a dati e parametri.

Per cogliere il vantaggio di questa formulazione è utile confrontarla con l'approccio precedente. I sistemi TTS tradizionali predicono rappresentazioni acustiche continue addestrando il modello a minimizzare l'errore rispetto al valore reale. Il problema è che per uno stesso testo esistono molte realizzazioni vocali ugualmente plausibili con diverse intonazioni, ritmi ed enfasi. Un modello che minimizza l'errore medio tende a produrre una media di tutte queste possibilità, con il risultato di un parlato appiattito e monotono: questo fenomeno è chiamato *over-smoothing*.

La formulazione a token discreti aggira questo limite poiché invece di predire un singolo valore continuo, il modello apprende una distribuzione di probabilità sul token successivo e ne campiona uno, esattamente come un modello linguistico sceglie la parola seguente. In questo modo non viene generata la media di tutte le voci plausibili, ma una specifica realizzazione coerente, recuperando l'espressività e la variabilità che l'over-smoothing sopprimeva. È questa la ragione di fondo per cui trattare l'audio come una sequenza di token, anziché come un segnale continuo da regredire, ha permesso un salto di naturalezza.

Il punto di partenza è la rappresentazione dell'audio in token introdotta da AudioLM, che per primo adotta una tokenizzazione su due livelli: i **token semantici**, che catturano la struttura a lungo termine e il contenuto dell'audio, e i **token acustici**, che ne preservano qualità, intonazione e timbro. Questa distinzione tra contenuto e dettaglio acustico è il filo conduttore che attraversa, in forme diverse, i sistemi descritti in seguito.

Nel contesto del TTS è SPEAR-TTS [17] ad applicare per primo la separazione semantico/acustico riformulando la sintesi come composizione di due compiti sequence-to-sequence: dal testo ai token semantici di alto livello (la fase di *reading*) e da questi ai token acustici di basso livello (la fase di *speaking*). Il vantaggio principale di questa separazione è la supervisione minima: il modulo speaking può essere addestrato su grandi quantità di audio non trascritto, mentre per la componente reading bastano pochi dati paralleli testo–audio. Per il controllo della voce, SPEAR-TTS adotta l'*example prompting*, generalizzando a parlanti non visti

a partire da un campione di pochi secondi, senza etichette o rappresentazioni esplicite del parlante.

Se SPEAR-TTS lavora sulla struttura della tokenizzazione, l'evoluzione successiva riguarda la scala. BASE TTS (Big Adaptive Streamable TTS with Emergent abilities) [18] è un sistema TTS su larga scala, multilingue e multi-speaker, addestrato su circa 100.000 ore di dati vocali di dominio pubblico. Tratta la sintesi come un problema di previsione del token successivo: un Transformer autoregressivo modella la distribuzione congiunta dei token di testo e dei codici vocali discreti prodotti da un codec, mentre un decoder convoluzionale ricostruisce la forma d'onda in modo incrementale e streamable. Questa formulazione consente di applicare direttamente al parlato i metodi sviluppati per i grandi modelli linguistici e attenua il problema dell'over-smoothing tipico dei primi sistemi TTS neurali, grazie alla natura probabilistica della predizione sui token. Gli autori mostrano inoltre che, all'aumentare di dati e parametri, emergono capacità prosodiche più sofisticate, in analogia con le abilità emergenti osservate negli Large Language Model (LLM).

Un'ulteriore evoluzione riguarda la natura dei token. Mentre nei sistemi visti finora i token vocali sono appresi in modo non supervisionato, con una conseguente mancanza di informazione semantica esplicita e di allineamento con il testo, CosyVoice [19] è il primo a integrare token vocali supervisionati. Il parlato è rappresentato con token semantici ottenuti da un modello di riconoscimento vocale multilingue, in cui viene inserita una quantizzazione vettoriale nell'encoder; a partire da questi token, la sintesi avviene tramite un LLM che genera i token dal testo e un modello di flow matching condizionale che li converte in voce. I risultati mostrano che i token semantici supervisionati superano nettamente quelli non supervisionati in termini di coerenza del contenuto e di somiglianza del parlante nella clonazione vocale zero-shot.

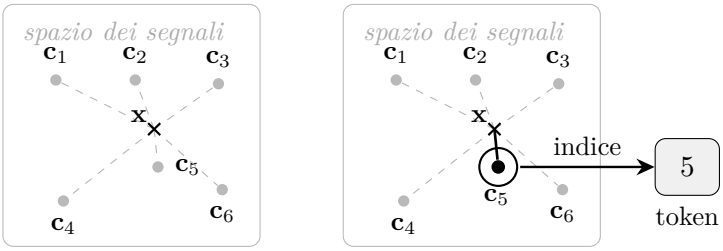
2.3 Speech tokenization e codec neurali audio

L'audio è un segnale continuo, mentre le tecniche di language modeling operano su sequenze di simboli discreti. Per applicarle al parlato è quindi necessaria una fase di discretizzazione, che trasformi il segnale continuo in una sequenza di unità finite. Lo strumento più diffuso a questo scopo è la **Vector Quantization (VQ)** [20].

La VQ si basa su un *codebook*, ovvero un dizionario di vettori di riferimento (*codeword*) ciascuno dei quali rappresenta una regione dello spazio dei possibili segnali. Il codebook viene appreso dai dati, tipicamente da algoritmi di clustering[3]

[3]Tecniche che raggruppano automaticamente i dati in insiemi (cluster) di elementi simili, senza etichette fornite a priori. Nel K-means, in particolare, ogni cluster è rappresentato dal proprio centroide: sono questi centroidi a costituire le codeword del codebook.

come il K-means, in modo che le codeword si collochino dove i dati sono più densi. Fissato il dizionario, ogni vettore in ingresso (ad esempio, la rappresentazione di un breve frame audio) viene confrontato con tutte le codeword e associato a quella più vicina secondo una metrica di distanza (solitamente quella euclidea). Il vettore continuo viene così sostituito dall'indice della codeword corrispondente, un numero intero che costituisce il token. (Figura 2.3)



(a) confronto (b) selezione

Figura 2.3: Vector Quantization (VQ). (a) Il vettore in ingresso $\mathbf{x}$ viene confrontato con tutte le codeword del codebook misurandone la distanza. (b) Si sceglie la codeword più vicina ($\mathbf{c}_5$, evidenziata): a $\mathbf{x}$ non si associa il vettore stesso, bensì l'*indice* che tale codeword occupa nel codebook, qui 5, che costituisce il token.

Quest'ultimo passaggio rende la VQ utile nelle applicazioni del TTS poiché ogni frame audio diventa un token discreto e l'intero segnale si riduce ad una sequenza di token, analoga ad una sequenza di parole in un testo. Su questa rappresentazione discreta si possono applicare le architetture dei modelli linguistici viste in precedenza.

La discretizzazione, tuttavia, non porta solo vantaggi: sostituire un vettore continuo con la codeword più vicina introduce un errore di quantizzazione, che può tradursi in distorsioni o artefatti nel segnale ricostruito. La qualità dipende fortemente da quanto il codebook è rappresentativo dei dati: codeword poco adatte aumentano l'errore e degradano la ricostruzione. In ambito audio, per coprire con un solo codebook l'intera varietà acustica del parlato, includendo timbri, intonazioni e fonemi, servirebbe un dizionario enorme, di fatto ingestibile sia in fase di addestramento che di inferenza.

Per superare questo limite è stata introdotta la **Residual Vector Quantization (RVQ)** [21]. Invece di affidarsi ad un singolo codebook ad altissima risoluzione, la RVQ impiega una cascata di codebook che lavorano in successione: il primo quantizza il vettore in ingresso fornendo un'approssimazione grossolana; il secondo non quantizza di nuovo il vettore originale, bensì l'errore residuo lasciato dal primo; il terzo quantizza l'errore residuo del secondo, e così via. A ogni stadio l'approssimazione viene raffinata, aggiungendo un livello di dettaglio in più. In questo modo si raggiunge un'alta fedeltà di ricostruzione mantenendo le dimensioni

di ogni singolo codebook contenute. Ed è proprio su questo principio che si basano i codec neurali moderni.

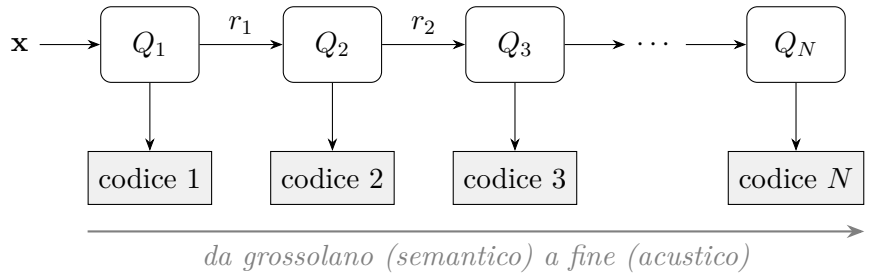


Figura 2.4: Residual Vector Quantization (RVQ): ogni stadio Q_i quantizza il residuo r_{i-1} lasciato dal precedente, producendo un codice. I primi codebook catturano la struttura portante, gli ultimi il dettaglio acustico.

I principi della VQ e della RVQ trovano la loro applicazione più matura nei codec neurali audio, architetture che imparano a comprimere e ricostruire la forma d'onda in modo interamente *data-driven*, ossia appreso direttamente dai dati anziché definito da regole esplicite. A differenza della VQ classica, il cui codebook viene costruito separatamente tramite clustering, in un codec neurale l'encoder, il quantizzatore e il decoder sono addestrati congiuntamente end-to-end [22], così che il codebook si specializzi sulle rappresentazioni effettivamente prodotte dalla rete.

Lo schema si articola in tre elementi principali. Un encoder convoluzionale trasforma la forma d'onda in una sequenza di vettori latenti a una frequenza di frame molto inferiore a quella di campionamento del segnale originale. Un modulo RVQ discretizza ciascun vettore latente in un insieme di codici, uno per ogni codebook della cascata. Infine, un decoder, speculare all'encoder, ricostruisce la forma d'onda a partire da questi codici. L'intero sistema viene ottimizzato per minimizzare l'errore di ricostruzione, tipicamente con l'ausilio di funzioni di perdita avversarie (GAN) che ne migliorano la qualità percettiva.

Il primo modello a coniugare questo schema con la quantizzazione residua è stato SoundStream [23], un codec interamente convoluzionale basato su RVQ, addestrato end-to-end e progettato per operare in streaming a bassa latenza. Sulla stessa linea si colloca EnCodec [24], che ne perfeziona l'addestramento e raggiunge una ricostruzione ad alta fedeltà, ed è stato adottato come tokenizer da diversi sistemi LLM-TTS, tra cui VALL-E. L'output di questi codec, ossia una sequenza di codici discreti, uno per codebook a ogni frame, costituisce esattamente i token acustici su cui operano i modelli discussi nella sezione precedente.

Una proprietà fondamentale della RVQ, che proviene direttamente dal suo funzionamento, è la gerarchia tra i codebook. Poiché ogni livello quantizza il residuo lasciato dal precedente, i codebook non hanno pari importanza: il primo

cattura la componente più significativa del segnale, mentre ciascun livello successivo aggiunge un contributo via via più fine. Ne risulta un ordinamento *coarse-to-fine*, in cui i primi livelli determinano la struttura portante della ricostruzione e gli ultimi ne definiscono i dettagli acustici più sottili, come le sfumature di timbro e prosodia.

Inoltre, proprio l'ordinamento gerarchico dei codebook rende possibile un compromesso regolabile tra qualità e costo. Poiché i livelli più fini contribuiscono solo al dettaglio, è possibile troncare la cascata conservando solo i primi k codebook degli N disponibili: si riduce così il numero di token prodotto per ogni frame, e quindi la quantità di informazione da generare e decodificare, al prezzo di una perdita di dettaglio acustico. Si configura così un trade-off diretto tra numero di codebook attivi e fedeltà del risultato.

In un codec RVQ "puro", tutti i codebook codificano informazione acustica e si distinguono soltanto per il livello di dettaglio; non esiste di per sé un codebook puramente semantico. La separazione tra contenuto semantico e dettaglio acustico è un concetto distinto, e alcuni sistemi progettano la tokenizzazione affinché il primo codebook si allinei al contenuto semantico e i successivi al dettaglio acustico.

2.4 Streaming e latenza nei sistemi TTS

Nei sistemi TTS la latenza rappresenta un aspetto cruciale. Se in contesti offline, come la produzione di audiolibri, il voiceover per contenuti video o la lettura di documenti, il tempo di generazione non costituisce un vincolo stringente, nelle applicazioni interattive (come assistenti vocali o agenti virtuali) la reattività è tanto importante quanto la qualità. Affinché l'interazione risulti naturale, l'utente deve percepire un ritardo minimo, paragonabile a quello di una conversazione reale.

Per rispondere a questa esigenza, i sistemi TTS possono operare secondo due modalità principali: *non-streaming* e *streaming*. Nella modalità non-streaming l'intera clip audio viene generata prima di avviarne la riproduzione: la latenza percepita coincide quindi con il tempo necessario all'inferenza. Al contrario, la modalità streaming si fonda su un meccanismo a blocchi: l'audio viene prodotto ed emesso in frammenti progressivi, chiamati *chunk*. Questo approccio permette di avviare la riproduzione non appena il primo blocco è pronto, mentre la generazione dei token successivi prosegue in parallelo, abbattendo la latenza percepita dall'utente.

L'efficacia e la stabilità di questa modalità vengono valutate attraverso specifiche metriche:

- **Latenza al primo chunk**: il tempo che intercorre tra il testo in input e l'emissione del primo blocco in output. È l'indicatore principale della reattività percepita.

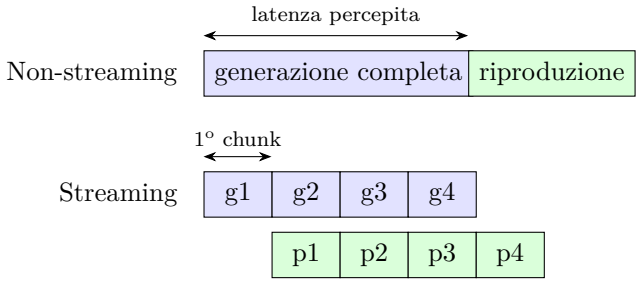


Figura 2.5: Modalità non-streaming e streaming

- **Real-Time Factor (RTF)**: il rapporto tra tempo di generazione e durata dell'audio prodotto. Un valore RTF < 1 indica una generazione più veloce del tempo reale, condizione che consente uno streaming continuo; un valore RTF > 1 indica una generazione più lenta del tempo reale.

- **Latenza inter-chunk**: il tempo di attesa tra la riproduzione di un chunk e il successivo. Se RTF > 1, il modello non riesce a tenere il passo con la riproduzione, causando l'interruzione del flusso audio con dei silenzi (*buffer underrun*).

Garantire un RTF < 1 e una bassa latenza al primo chunk rappresenta una sfida architetturale notevole, specialmente per i modelli basati sulla generazione autoregressiva. In questi sistemi, poiché ogni token viene prodotto condizionandolo ai precedenti, l'inferenza è strettamente sequenziale e non parallelizzabile lungo l'asse temporale. Ne consegue che il tempo di generazione cresce linearmente all'aumentare della lunghezza del testo in input.

In questo scenario, la frequenza di campionamento (*frame-rate*) del tokenizer assume un ruolo determinante. Poiché la generazione procede un token alla volta, il numero di passi autoregressivi necessari per sintetizzare una determinata durata di audio dipende direttamente dal frame-rate: una frequenza più bassa comporta un minor numero di token e, di conseguenza, un abbattimento dei tempi di latenza.

Oltre all'architettura del modello, anche la configurazione del meccanismo a blocchi richiede un delicato bilanciamento. La dimensione dei chunk influisce direttamente sull'esperienza utente: chunk di piccole dimensioni minimizzano la latenza iniziale, ma aumentano l'overhead computazionale e il rischio di buffer underrun, lasciando un margine di tempo ristretto per la generazione del blocco successivo. Viceversa, chunk di dimensioni maggiori garantiscono un flusso più stabile e continuo, al prezzo di un'attesa iniziale più lunga.

Per ottimizzare questi parametri e contenere la latenza complessiva, la letteratura ha sviluppato diverse strategie. A livello architetturale, si privilegiano modelli

parzialmente parallelizzati e codec a bassa frequenza. Dal punto di vista dell'efficienza computazionale, la latenza di inferenza può essere ridotta sia mediante tecniche di ottimizzazione del modello, quali la distillazione e la quantizzazione dei pesi, sia attraverso ottimizzazioni del processo di inferenza, come il riuso delle computazioni intermedie (*Key-Value (KV) caching*) e l'esportazione del modello in formati ottimizzati per il deployment. Infine, l'impiego di *vocoder* neurali non-autoregressivi ed efficienti previene che lo stadio finale di ricostruzione della forma d'onda diventi un collo di bottiglia.

2.5 Inferenza Machine Learning in Unity: ONNX e Sentis

Le sezioni precedenti hanno illustrato l'evoluzione dei modelli TTS e i principi su cui si fonda la generazione del parlato. Affinché un modello di questo tipo possa essere effettivamente integrato in un'applicazione interattiva, è necessario affrontare il problema del passaggio dall'ambiente di addestramento a quello di esecuzione.

I modelli di deep learning vengono tipicamente sviluppati in framework come PyTorch o TensorFlow, in ambienti basati su Python e supportati da acceleratori hardware dedicati. Sebbene questo contesto sia ottimale per la fase di addestramento, si scontra con le esigenze della fase di inferenza all'interno di un'applicazione reale. In questo scenario, il modello, con i parametri ormai fissati, deve trasformare gli input in output nel modo più efficiente possibile, operando direttamente sulle risorse di calcolo della macchina ospite, le quali presentano spesso stringenti vincoli di memoria e potenza.

Si pone quindi la necessità di portare un modello al di fuori del framework di origine per eseguirlo in un ambiente di inferenza efficiente e indipendente da Python. È in questo contesto che si collocano i due strumenti descritti in seguito: un formato standard per rappresentare il modello (ONNX) e un runtime in grado di eseguirlo direttamente nel motore Unity (Sentis).

Il primo strumento è ONNX [25], un formato standard e aperto per la rappresentazione di modelli di deep learning, nato con l'obiettivo di disaccoppiare la fase di sviluppo da quella di deployment. Un modello ONNX è rappresentato internamente come un grafo computazionale: i nodi corrispondono alle operazioni (convoluzioni, moltiplicazioni matriciali, funzioni di attivazione, e così via), mentre gli archi rappresentano i tensori che fluiscono tra di esse. Ogni operazione appartiene ad un insieme standardizzato di operatori definito dalla specifica ONNX: ciò garantisce che un modello esportato sia interpretabile in modo coerente da qualunque runtime conforme allo standard.

Per eseguire tale grafo all'interno di Unity, si è fatto ricorso a Unity Sentis [26], la libreria ufficiale del motore per l'inferenza di reti neurali. Sentis consente di

importare un modello ONNX come asset e di eseguirlo in tempo reale, sfruttando l'hardware del dispositivo locale. Il flusso di lavoro prevede che il modello caricato in memoria venga gestito da un componente chiamato worker, il quale traduce il grafo in una sequenza di operazioni, le pianifica sull'hardware selezionato e restituisce i tensori di output.

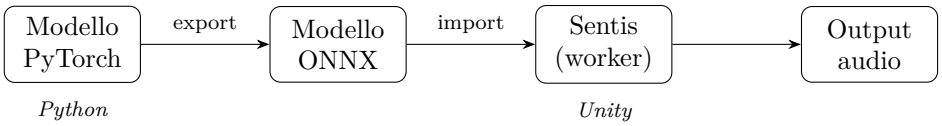


Figura 2.6: Dal modello addestrato all'esecuzione nativa: il modello PyTorch viene esportato in ONNX e caricato in Unity Sentis, che ne esegue l'inferenza in locale.

A differenza delle soluzioni cloud-based, Sentis esegue il modello localmente e all'interno del motore stesso: l'inferenza diventa così parte integrante dell'applicazione, senza dipendere da una connessione di rete o da un server remoto. Questo aspetto è centrale per gli obiettivi del presente lavoro, in cui si richiede che l'intera pipeline TTS risieda e operi nativamente in Unity.

L'esecuzione di un modello tramite Sentis è tuttavia soggetta ad alcuni vincoli, che derivano dal dover far funzionare in un game engine, modelli concepiti per ambienti di calcolo più flessibili. Una prima limitazione riguarda la copertura degli operatori: Sentis supporta i modelli ONNX entro un determinato intervallo di versioni del set di operatori (opset) e non tutte le operazioni definite dallo standard ONNX sono implementate. Un modello che faccia uso di operatori non supportati non può quindi essere eseguito direttamente e richiede interventi di adattamento, come la sostituzione o la riscrittura delle operazioni problematiche in fase di esportazione.

Un secondo aspetto critico riguarda la gestione delle forme dei tensori. I modelli che operano su sequenze di lunghezza variabile, come quelli generativi autoregressivi, producono grafi con dimensioni dinamiche o simboliche, ossia non note a priori. La loro corretta interpretazione da parte del runtime non è sempre immediata e può richiedere di esplicitare o vincolare tali dimensioni perché l'inferenza proceda correttamente.

Un terzo aspetto riguarda l'efficienza dell'esecuzione su dispositivo. Poiché l'inferenza avviene sulle risorse locali, spesso più limitate di quelle di un server, è prassi ridurre l'occupazione di memoria e il costo computazionale del modello attraverso tecniche di compressione. Tra queste, la quantizzazione, ossia la rappresentazione dei pesi con una precisione numerica ridotta, consente di alleggerire il modello con un impatto contenuto sulla qualità.

Questi vincoli definiscono il quadro entro cui deve essere progettato l'export di un modello TTS verso Sentis.

La scelta di Sentis come runtime di inferenza è motivata dal requisito di mantenere l'intera pipeline TTS nativa all'interno di Unity. Esistono infatti approcci alternativi per eseguire modelli neurali in un'applicazione di questo tipo. Una prima possibilità consiste nell'affidare l'inferenza a un servizio remoto, inviando il testo a un server che restituisce l'audio sintetizzato: una soluzione che però reintroduce proprio le dipendenze, come connessione di rete, latenza variabile, costi e gestione della privacy dei dati, che questo lavoro intende evitare. Una seconda possibilità è il ricorso a runtime esterni integrati tramite plugin nativi, come ONNX Runtime [27]; è la via seguita, ad esempio, da progetti come Orpheus-TTS-Unity [28], che esegue un modello TTS in Unity appoggiandosi a librerie di inferenza esterne.

Rispetto a queste alternative, Sentis offre il vantaggio di essere integrato direttamente nel motore: non richiede componenti esterni al di fuori del pacchetto ufficiale, opera su tutte le piattaforme supportate da Unity e mantiene l'inferenza all'interno del ciclo di esecuzione dell'applicazione.

2.6 Sistemi TTS allo stato dell'arte

Dopo aver introdotto i principi e l'evoluzione dei sistemi TTS neurali, questa sezione presenta una rassegna dei principali sistemi di sintesi vocale oggi disponibili. Il panorama attuale può essere suddiviso in due grandi categorie: da un lato i sistemi commerciali proprietari, offerti come servizi cloud; dall'altro i sistemi open source, che negli ultimi anni hanno raggiunto livelli di naturalezza comparabili.

2.6.1 Sistemi commerciali

Lo stato dell'arte in termini di qualità e naturalezza del parlato sintetico è oggi rappresentato in larga parte da sistemi commerciali proprietari, offerti come servizi cloud accessibili tramite API. Tra questi, ElevenLabs [29] è uno dei riferimenti più riconosciuti, apprezzato per la naturalezza, l'espressività e in particolare per il controllo esplicito dell'emozione e le capacità di clonazione vocale.

Accanto a esso, i principali fornitori di servizi cloud mettono a disposizione soluzioni TTS mature e ampiamente diffuse, come Azure AI Speech di Microsoft [30], Google Cloud Text-to-Speech [31] e Amazon Polly [32], caratterizzate da un ampio supporto multilingue e da una stretta integrazione con i rispettivi ecosistemi; più recentemente si è aggiunta anche l'offerta di OpenAI [33].

Questi sistemi condividono un modello di funzionamento comune: l'inferenza avviene su server remoti e l'applicazione client si limita a inviare il testo e a ricevere l'audio sintetizzato. Tale architettura offre qualità elevata e solleva l'applicazione dall'onere computazionale della generazione, ma vincola il loro utilizzo

alla disponibilità di un servizio esterno, essendo i modelli chiusi e non eseguibili localmente.

2.6.2 Open source

Sul fronte opposto, i sistemi open source hanno colmato gran parte del divario qualitativo che li separava dalle soluzioni commerciali, arrivando in alcuni confronti di preferenza soggettiva a risultare competitivi con esse. A differenza dei servizi proprietari, questi sistemi rendono disponibili pesi e codice, permettendo l'esecuzione locale, l'ispezione del modello e l'adattamento a domini specifici. Si distinguono però per approcci architetturali eterogenei, in particolare per il modo in cui rappresentano l'audio. Di seguito si descrivono i quattro modelli open source considerati in questo lavoro come oggetto della valutazione soggettiva, scelti in modo da coprire i principali paradigmi di sintesi discussi nelle sezioni precedenti e, al contempo, per il loro supporto multilingue, requisito necessario alla generazione di parlato in italiano.

Il primo è Qwen3-TTS [34], sviluppato da Alibaba e rilasciato con licenza permissiva (Apache 2.0). Si tratta di un sistema LLM-TTS basato su token audio discreti prodotti da uno stack di quantizzazione vettoriale residua, in grado di clonare una voce a partire da pochi secondi di riferimento. È il modello al centro di questo lavoro e quello integrato nella pipeline nativa in Unity; la sua architettura è descritta in dettaglio nel Capitolo 3.

Allo stesso paradigma a token discreti appartiene OmniVoice [35], un modello zero-shot massivamente multilingue. A differenza dei sistemi a due stadi, OmniVoice adotta un'architettura non-autoregressiva di tipo diffusion language model che mappa il testo direttamente in token acustici multi-codebook, in un unico stadio. Si distingue per una copertura linguistica molto ampia e per l'efficienza derivante dalla generazione non-autoregressiva.

Un approccio differente è rappresentato da VoxCPM [36], sviluppato da OpenBMB. VoxCPM è un sistema *tokenizer-free*: invece di discretizzare l'audio in token, genera direttamente rappresentazioni continue del parlato attraverso un'architettura diffusion-autoregressiva, evitando del tutto lo stadio di tokenizzazione. Costituisce quindi il controesempio rispetto ai sistemi basati su token discreti, ed è caratterizzato da una sintesi particolarmente naturale e sensibile al contesto.

Infine, Chatterbox [37], sviluppato da Resemble AI e rilasciato con licenza MIT, è un modello multilingue (italiano incluso) basato su un backbone Transformer di tipo Llama, che offre clonazione vocale zero-shot. Dispone inoltre di un parametro di controllo dell'intensità espressiva (*exaggeration*), sebbene nel presente lavoro tutti i modelli siano impiegati nella sola modalità di clonazione da riferimento, per garantire l'uniformità del confronto.

2.6.3 Analisi comparativa e gap

Alla luce della rassegna presentata, i sistemi TTS finora descritti possono essere valutati lungo alcune dimensioni critiche, determinanti per stabilirne l'idoneità rispetto agli obiettivi di questo lavoro.

La prima di queste è la qualità e naturalezza del parlato. Sotto questo profilo, il divario storico tra soluzioni proprietarie e aperte si è in larga parte colmato: i migliori sistemi open source odierni raggiungono livelli di naturalezza paragonabili a quelli commerciali, risultando altamente competitivi, in diversi test di preferenza soggettiva, persino con riferimenti del calibro di ElevenLabs.

Una seconda dimensione, di importanza centrale per il dominio applicativo di questa tesi, è l'espressività emotiva nella clonazione vocale. Tale metrica definisce la capacità di un sistema di riprodurre fedelmente non soltanto il timbro, ma anche la carica emotiva e l'intonazione caratteristiche dell'audio di riferimento.

Un terzo aspetto cruciale riguarda la latenza e il supporto allo streaming, ovvero la capacità di generare l'audio in modo sufficientemente rapido e incrementale da garantirne l'usabilità in scenari interattivi real-time. A queste dimensioni prestazionali si affiancano fattori pratici quali la copertura linguistica, con particolare attenzione al supporto per la lingua italiana, e il modello di licenza. Quest'ultimo detta le condizioni di utilizzo e modifica: se i sistemi commerciali impongono un approccio chiuso erogato come servizio, l'ecosistema open source spazia da licenze permissive (come Apache 2.0 e MIT) a vincoli più restrittivi per l'uso commerciale.

Tuttavia, la dimensione maggiormente discriminante ai fini di questo progetto è la modalità di esecuzione. Le soluzioni commerciali operano esclusivamente in cloud, portando con sé limitazioni legate alla dipendenza dalla rete, ai costi operativi e alla privacy dei dati. Di contro, i sistemi open source, pur supportando l'esecuzione locale, nascono nativamente per ambienti Python e per architetture hardware di classe server o workstation. In nessuno dei due scenari è prevista un'integrazione diretta all'interno di un motore di rendering che consenta l'esecuzione on-device in assenza di dipendenze esterne.

Questo quadro evidenzia un'ulteriore e significativa lacuna tecnologica sul fronte dell'espressività: allo stato attuale, per la lingua italiana, nessun sistema coniuga la disponibilità open source con un controllo emotivo esplicito (come quello offerto dai tag proprietari di ElevenLabs). I modelli aperti, infatti, affidano l'espressività italiana prevalentemente al trasferimento dell'emozione tramite clonazione, mancando di meccanismi di controllo granulare altrettanto maturi. Per questa ragione, la valutazione soggettiva presentata nei capitoli successivi misurerà l'espressività dei modelli analizzando proprio la loro efficacia nella clonazione a partire da riferimenti puramente emotivi.

Da tale analisi emerge con chiarezza lo spazio di ricerca in cui si colloca il presente lavoro. Se da un lato i sistemi dotati della qualità necessaria non risultano

nativamente integrabili in applicazioni interattive, dall'altro l'incorporazione di un TTS neurale espressivo all'interno di un motore come Unity, con esecuzione rigorosamente locale, a bassa latenza e con supporto allo streaming, rappresenta una frontiera ancora poco esplorata. La vera sfida ingegneristica non si limita alla semplice inferenza on-device, bensì al mantenimento di un'espressività emotiva coerente e credibile, anche a valle delle operazioni di compressione e adattamento richieste per l'esecuzione su un dispositivo client.

Capitolo 3

Architettura di Qwen3-TTS

Tra i sistemi TTS open-source analizzati, la scelta per la realizzazione della pipeline nativa è ricaduta su **Qwen3-TTS**. Il modello è disponibile in più configurazioni, che si distinguono per la frequenza del tokenizer audio (25 Hz o 12,5 Hz) e per la dimensione (0,6 o 1,7 miliardi di parametri).

In questo lavoro è stata adottata la variante **12Hz-0.6B**. Questa decisione è stata guidata da quattro fattori principali.

1. **Qualità della sintesi in italiano.** Sull'italiano, Qwen3-TTS raggiunge una somiglianza del parlante (*speaker similarity*) paragonabile o superiore a quella dei sistemi commerciali di riferimento MiniMax [38] ed ElevenLabs, con un tasso di errore sul contenuto (*Word Error Rate*) anch'esso comparabile (Tabella 3.1) [34], a conferma della sua idoneità per il dominio applicativo di questo lavoro.

2. **Predisposizione alla bassa latenza.** La variante a 12,5 Hz è progettata per la generazione in streaming: operando su un numero ridotto di token e impiegando un decoder leggero, privo dei componenti più onerosi della variante a 25 Hz, consente di generare il primo segmento audio in tempi molto rapidi, requisito importante per un'esecuzione locale e interattiva.

3. **Esecuzione on-device.** È stata privilegiata la variante da 0,6 miliardi di parametri, anziché quella da 1,7, per rispettare i vincoli di memoria e di calcolo di un'esecuzione in locale, pur a fronte di una qualità leggermente inferiore ma comunque adeguata allo scopo.

4. **Idoneità all'integrazione nativa.** Il sistema presenta una struttura modulare ed è rilasciato con licenza permissiva (Apache 2.0), che ne garantisce flessibilità d'impiego e libertà di modifica.

Tabella 3.1: Confronto sulla lingua italiana tra la variante impiegata (Qwen3-TTS-12Hz-0.6B), la variante maggiore e i sistemi commerciali di riferimento. *Word Error Rate (WER)*: valori più bassi migliori; *speaker similarity*: valori più alti migliori. Dati dal technical report [34].

Metrica (italiano)	**12Hz-0.6B**	**12Hz-1.7B**	**MiniMax**	**ElevenLabs**
WER ↓	1,534	**0,948**	1,543	1,743
Speaker Similarity ↑	0,792	**0,817**	0,699	0,579

Il presente capitolo analizza nel dettaglio l'architettura di questo modello. Ai fini di questo studio sono state impiegate due declinazioni del sistema, che condividono la medesima infrastruttura di base e si distinguono unicamente per la modalità di definizione della voce:

- Variante *Base*: permette la clonazione vocale zero-shot estraendo le caratteristiche timbriche da un breve file audio di riferimento.
- Variante *CustomVoice*: genera una voce predefinita, le cui caratteristiche sono state apprese direttamente in fase di addestramento.

I dettagli architetturali di seguito riportati sono tratti dal technical report [34]. I parametri di configurazione non esplicitamente documentati nel paper sono stati ricavati dall'ispezione diretta del codice sorgente del repository ufficiale[1].

Il sistema elabora l'audio operando in sequenza attraverso tre moduli principali:

- **LM Backbone (Talker)**: il modello linguistico di base incaricato della modellazione acustico-linguistica.
- **Multi-Token Prediction (MTP)**: un modello autoregressivo che, a partire dall'output del backbone, genera i restanti codebook acustici residui di ciascun frame.
- **Code2Wav**: il decoder neurale che ricostruisce la forma d'onda continua a partire dalla rappresentazione discreta.

A questa pipeline si affianca il **tokenizer audio** a 12,5 Hz, responsabile della conversione bidirezionale tra la forma d'onda e i token discreti. Infine, per abilitare il voice cloning, la variante Base integra un ulteriore componente: lo **Speaker Encoder**. Questo modulo analizza la reference vocale e ne estrae l'*embedding*, ossia

[1] `https://github.com/QwenLM/Qwen3-TTS`

un vettore denso che ne racchiude le caratteristiche acustiche in forma numerica compatta, per poi condizionare la generazione vocale.

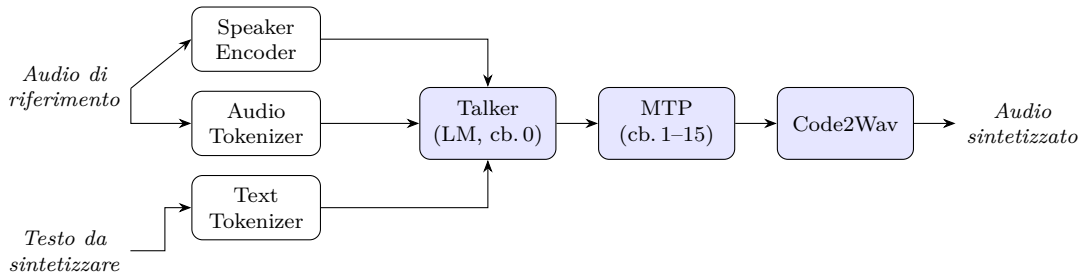


Figura 3.1: Architettura di Qwen3-TTS-12Hz.

3.1 Tokenizer

Il tokenizer audio impiegato dalla variante a 12,5 Hz, denominato **Qwen-TTS-Tokenizer-12Hz**, ha il compito di convertire il segnale audio continuo in una sequenza di codici discreti e, in fase di decodifica, di ricostruire la forma d'onda a partire da essi. Questo componente riprende la strategia di quantizzazione a flussi semantico e acustico separati, ispirata al codec neurale Mimi [39] del sistema Moshi, applicando il principio della quantizzazione vettoriale residua (RVQ) (Sezione 2.3). Nello specifico, ogni frame audio è rappresentato a 12,5 Hz da un **codebook semantico** e da un **modulo RVQ a 15 livelli** che ne modella il dettaglio acustico, per un totale di 16 codici per frame [34]. Questo frame rate volutamente basso riduce il numero di token da generare, favorendo una bassa latenza.

Coerentemente con la gerarchia *coarse-to-fine* propria della RVQ, i codebook assumono ruoli distinti. Il primo codebook codifica il contenuto semantico del parlato: durante l'addestramento, questo livello è supervisionato da WavLM [40], un modello pre-addestrato per la rappresentazione auto-supervisionata del parlato che agisce da *teacher*, in modo da allineare le rappresentazioni alla struttura linguistica. I successivi quindici codebook residui modellano invece i dettagli acustici (timbro, prosodia e intonazione), raffinando progressivamente la ricostruzione. Questa separazione tra contenuto semantico e dettaglio acustico è il principio che consente al backbone del sistema di occuparsi esclusivamente della generazione del codebook semantico, affidando i restanti al modulo dedicato.

Il tokenizer è addestrato con un framework di tipo GAN: il generatore opera direttamente sulla forma d'onda grezza per estrarre e quantizzare le due rappresentazioni, mentre il discriminatore ne valuta l'output per migliorare la naturalezza e la fedeltà.

Un'ulteriore caratteristica è che encoder e decoder sono **completamente causali**, ossia non impiegano alcun contesto futuro (*look-ahead*): ciò abilita la decodifica in streaming con latenza minima, rendendo possibile l'emissione del primo segmento audio già a partire dai primi frame generati.

3.2 LM Backbone (Talker)

Il **LM backbone**, denominato **Talker**, costituisce il modulo centrale del sistema. Si tratta di un Language Model autoregressivo basato sull'architettura Qwen3, attorno al quale si articolano tutti gli altri componenti. A differenza dei tradizionali modelli linguistici testuali, il backbone è addestrato per prevedere il token audio successivo anziché la parola successiva: esso genera sequenzialmente i token del codebook semantico, ovvero il *codebook 0*, condizionando ogni previsione sull'intera sequenza pregressa. La generazione dei codebook acustici residui è invece a carico del modulo MTP (Sezione 3.3).

Poiché il technical report ufficiale non dettaglia i parametri architetturali interni, i valori discussi in questa sezione (riepilogati nella Tabella 3.2) sono stati ricavati mediante l'ispezione diretta dei file di configurazione e del codice sorgente. Il backbone è implementato come un Transformer da circa 440,5 milioni di parametri, strutturato in 28 layer con una *hidden size* di 1024 (ovvero la dimensione dei vettori con cui ciascun token è rappresentato internamente al modello). Ogni layer adotta l'architettura consolidata dei Transformer moderni: un meccanismo di *self-attention* seguito da una rete *feed-forward*, con normalizzazione RMSNorm e connessioni residue. L'informazione posizionale dei token è gestita tramite Rotary Position Embedding (RoPE) [41], un approccio che codifica la posizione applicando rotazioni agli embedding anziché sommare vettori posizionali assoluti. Tale meccanismo risulta particolarmente efficace per estendere il contesto utile, consentendo di gestire sequenze lunghe fino a 32 768 token.

Nella sua implementazione concreta, il backbone non impiega il RoPE nella forma standard appena descritta, ma una sua estensione multidimensionale, denominata Multimodal Rotary Position Embedding (M-RoPE). Questo meccanismo era stato introdotto per Qwen2-VL allo scopo di codificare, in sequenze video, posizioni definite lungo più assi indipendenti: uno temporale e due spaziali, corrispondenti all'altezza e alla larghezza del fotogramma. A ciascuno di questi assi il M-RoPE applica una rotazione distinta, ripartendo tra loro le componenti di frequenza del RoPE.

Nel caso di Qwen3-TTS, però, questa capacità multidimensionale non trova reale impiego: l'ingresso è costituito unicamente da testo e audio, privo della struttura spaziale per cui il meccanismo era concepito. I tre indici posizionali associati

a un token assumono sempre lo stesso valore[2] e il M-RoPE degenera, in questo caso d'uso, in un ordinario RoPE a un solo asse. Il backbone eredita dunque l'infrastruttura multidimensionale dai modelli multimodali della famiglia Qwen3 senza che la dimensionalità aggiuntiva svolga un ruolo effettivo. Questo dettaglio implementativo resta comunque rilevante in fase di esportazione (Sezione 4.3.1): il grafo ONNX richiede infatti che le posizioni siano fornite nell'assetto a tre assi, anche se i tre valori coincidono.

Sotto il profilo dell'efficienza in fase di inferenza, risulta cruciale l'adozione della Grouped Query Attention (GQA). Per comprendere il vantaggio della GQA è utile richiamare il funzionamento del meccanismo di *self-attention*. Nell'elaborare una sequenza, il modello mette in relazione ogni token con tutti gli altri per stabilire il rispettivo grado di rilevanza concettuale. Questa relazione è calcolata attraverso tre vettori generati da ciascun token: un *query* (ciò che il token "cerca"), una *key* (ciò che il token "offre") e un *value* (l'informazione che il token trasporta). Confrontando la query di un token con le key di tutti gli altri, il modello determina a quali elementi prestare attenzione e ne aggrega di conseguenza i relativi value. Per cogliere relazioni eterogenee in parallelo, questo calcolo viene eseguito simultaneamente da più *teste* di attenzione indipendenti (*multi-head attention*), ciascuna dotata dei propri vettori query, key e value.

Questo schema comporta tuttavia un costo computazionale rilevante in fase di generazione. Poiché il modello produce un token alla volta condizionandosi sulla sequenza passata, i vettori *key* e *value* già calcolati vengono mantenuti in memoria per evitare ridondanze nei passi successivi: tale struttura costituisce la cosiddetta **KV cache**. La sua dimensione cresce linearmente con la lunghezza della sequenza e con il numero di teste, diventando un fattore critico per l'esecuzione dei modelli su dispositivi dalle risorse limitate.

È in questo contesto che interviene la GQA [42]. Invece di assegnare a ogni testa di query una coppia indipendente di key e value, questa tecnica raggruppa le teste di query affinché condividano lo stesso insieme di chiavi e valori. Nel backbone esaminato, le 16 teste di query sono configurate per interfacciarsi con sole 8 teste di chiave-valore (con un rapporto di 2:1). Questo approccio dimezza la dimensione della KV cache e riduce drasticamente l'occupazione di memoria durante l'inferenza, fornendo un vantaggio diretto per l'esecuzione locale.

Prima di entrare nel backbone, i token testuali e quelli audio devono essere convertiti in vettori. I due tipi partono però da spazi di dimensione diversa, e appositi

[2] "the three rotary position index [...] is always the same, so the text embedding rotary position embedding has no difference with modern LLMs".

moduli di embedding li portano entrambi nello spazio interno del Transformer.

Il **text embedding** converte ciascun token testuale, estratto da un vocabolario di 151 936 elementi, in un vettore di dimensione 2048; poiché tale dimensionalità differisce dalla hidden size del backbone, un modulo di proiezione (*text projection*) lo riporta a 1024 componenti. Il **codec embedding** mappa invece ciascun token del codebook 0 (vocabolario di 3072 elementi) direttamente in un vettore di dimensione 1024, così da reintrodurre i token audio già generati come contesto autoregressivo. Entrambi i flussi, testuale e acustico, convergono dunque nello stesso spazio a 1024 dimensioni.

Resta da precisare come questi due flussi vengano combinati in ciascuna posizione della sequenza. Il technical report descrive l'approccio *dual-track* come una concatenazione lungo l'asse dei canali; l'ispezione del codice sorgente rivela però che l'implementazione effettiva ricorre a una **somma posizionale**: i due vettori vengono sommati elemento per elemento, preservando la dimensione 1024. Una concatenazione, al contrario, produrrebbe vettori di dimensione 2048, incompatibili con il backbone in assenza di un ulteriore livello di proiezione, che nel codice non è presente. Questa discrepanza, verificata sia sul codice sorgente sia tramite analisi numerica degli embedding (Appendice A), è emersa nel corso dell'analisi condotta per l'esportazione della pipeline.

Tabella 3.2: Specifiche architetturali del LM backbone (Talker) della variante Qwen3-TTS-12Hz-0.6B. I valori sono stati ricavati dall'ispezione della configurazione e del codice sorgente del modello.

Parametro	**Valore**
Numero di layer Transformer	28
Dimensione nascosta (*hidden size*)	1024
Dimensione rete *feed-forward*	3072
Attivazione *feed-forward*	SiLU
Normalizzazione	RMSNorm ($\epsilon = 10^{-6}$)
Tipo di attenzione	GQA (16 teste Q / 8 teste KV)
Dimensione per testa	128
Codifica posizionale	M-RoPE($\theta = 10^6$)
Lunghezza massima di contesto	32 768 token
Vocabolario testuale	151 936
Dimensione text embedding	2048
Vocabolario codec (*codebook 0*)	3072
Parametri del backbone	440,5 M

3.3 Modulo MTP

Il backbone genera unicamente il codebook semantico; i quindici codebook acustici residui sono prodotti dal modulo MTP [34]. Il technical report ne descrive il ruolo: generare i codebook residui a partire dall'output del Talker, abilitando la decodifica audio già a partire dal primo frame codec e riducendo così la latenza complessiva. Tuttavia non specifica né l'architettura interna né il meccanismo implementativo.

L'analisi del codice sorgente ha rivelato che il MTP è un **secondo Language Model causale**, distinto dal backbone e nel codice denominato *Code Predictor*[3]. Con circa 141,6 milioni di parametri, ne condivide l'impianto architetturale (attenzione di tipo GQA con 16 teste di query e 8 di chiave-valore, dimensione nascosta 1024) ma è più compatto, essendo composto da soli 5 layer Transformer contro i 28 del backbone.

La differenza più rilevante riguarda le teste di uscita, ossia gli strati finali che convertono lo stato interno del modello nella distribuzione di probabilità sul vocabolario da cui viene campionato il token. Mentre il backbone dispone di un'unica testa di uscita, dedicata al codebook 0, il MTP ne possiede **quindici**, una per ciascun codebook residuo, ciascuna operante su un vocabolario codec di dimensione 2048. Le specifiche sono riepilogate nella Tabella 3.3.

A partire dall'ultimo stato nascosto del backbone e dall'embedding del codebook 0 appena generato, il Code Predictor produce in sequenza i token dei codebook da 1 a 15 relativi a quel frame. Ciò comporta una conseguenza implementativa rilevante: la generazione di un singolo frame audio non richiede una sola inferenza, ma un **ciclo annidato**. A ogni passo del ciclo principale del backbone corrisponde infatti un secondo ciclo autoregressivo interno, quello del MTP, che itera sui quindici codebook residui.

3.4 Code2Wav

L'ultimo stadio della pipeline è il Code2Wav, il decoder incaricato di ricostruire la forma d'onda a partire dai sedici codebook RVQ generati in precedenza dal backbone e dal modulo MTP. Il technical report lo descrive come una *rete convoluzionale causale leggera*: elaborando ogni frame sulla base del solo contesto passato, senza mai aspettare i campioni futuri, il decoder può iniziare a produrre audio via via che i codici vengono generati, senza dover disporre dell'intera sequenza. È questa natura puramente convoluzionale e causale a consentire la decodifica in streaming a bassa latenza.

[3]Nel codice sorgente il modulo corrisponde alla classe `Qwen3TTSCodePredictorForCausalLM`.

Tabella 3.3: Specifiche architetturali del modulo MTP (Code Predictor) della variante Qwen3-TTS-12Hz-0.6B. Valori ricavati dall'ispezione della configurazione e del codice sorgente del modello.

Parametro	**Valore**
Numero di layer Transformer	5
Dimensione nascosta (*hidden size*)	1024
Tipo di attenzione	GQA (16 teste Q / 8 teste KV)
Numero di teste di uscita (*LM head*)	15 (codebook 1–15)
Vocabolario codec (per codebook)	2048
Parametri del modulo	141,6 M

Secondo il report, la ricchezza della rappresentazione multi-codebook a 12,5 Hz, in cui ogni frame condensa sia il contenuto semantico sia i dettagli acustici, rende superflui due componenti impiegati invece dalla variante a 25 Hz: l'estrazione di un vettore del parlante ausiliario e il ricorso a modelli diffusivi complessi. Quest'ultima variante, operando su una rappresentazione a codebook singolo e quindi meno densa, necessita di meccanismi più onerosi per raggiungere una qualità di ricostruzione equivalente.

La struttura del decoder, non documentata nel report, è stata ricostruita dall'analisi del grafo ONNX esportato e del relativo codice sorgente[4].

Il Code2Wav, che conta circa 114,3 milioni di parametri, riceve i sedici codici di ciascun frame e li converte in un segnale PCM a 24 kHz tramite una successione di stadi. Il primo è uno *Split Residual Vector Quantizer*, che ricostruisce la rappresentazione latente: i codebook sono suddivisi in due gruppi, quello semantico (codebook 0) e quello acustico (codebook 1–15), decodificati separatamente e infine sommati, coerentemente con la separazione semantico/acustico propria del tokenizer. La rappresentazione così ottenuta attraversa quindi un blocco convoluzionale causale, un modulo Transformer e una fase di *upsampling* temporale, prima di essere elaborata da una cascata finale di blocchi convoluzionali che ne espande progressivamente la risoluzione fino alla forma d'onda.

Una caratteristica architetturale rilevante, emersa dall'analisi del grafo, è l'impiego della funzione di attivazione **SnakeBeta**, definita come

$$x + \frac{1}{e^{\beta} + \epsilon} \sin^2(x\, e^{\alpha}) \,,$$

[4]Classe `Qwen3TTSTokenizerV2Decoder` nel modulo del tokenizer.

una funzione periodica caratteristica delle architetture di tipo DAC [43], non citata esplicitamente nel report.

3.5 Speaker Encoder

Lo **speaker encoder** è il componente responsabile della cattura dell'identità vocale nella modalità di clonazione, ed è pertanto presente unicamente nella variante Base[5].

Il technical report indica che si tratta di un encoder del parlante *learnable*, ossia addestrato congiuntamente al resto del sistema; l'ispezione del codice sorgente ne rivela l'architettura, basata sul modello **ECAPA-TDNN** [44], ampiamente adottato nei sistemi di verifica del parlante. A differenza degli approcci che impiegano estrattori di embedding pre-addestrati e separati, l'addestramento congiunto allinea le rappresentazioni prodotte dall'encoder allo spazio interno del modello generativo.

L'encoder riceve in ingresso le feature mel (la rappresentazione tempo-frequenza) dell'audio di riferimento (128 bande) e le elabora attraverso una successione di blocchi convoluzionali di tipo *time-delay* (TDNN) e Res2Net con meccanismi di attenzione sui canali. Un modulo di *attentive statistics pooling* aggrega quindi l'informazione lungo l'asse temporale, calcolando statistiche (media e deviazione standard) pesate sull'intera sequenza, così da condensare un audio di durata variabile in un unico vettore di dimensione fissa. Una proiezione finale produce l'**embedding del parlante**, un vettore di dimensione 1024 che ne sintetizza l'identità vocale (timbro, registro e caratteristiche prosodiche). Il modulo, di dimensioni contenute (circa 8,9 milioni di parametri), è così in grado di rappresentare una voce a partire da pochi secondi di riferimento.

Tale embedding viene inserito in una posizione fissa della sequenza di ingresso del backbone, dove condiziona l'intera generazione: da quel momento ogni token audio prodotto riflette la voce di riferimento, senza che i moduli a valle debbano accedere nuovamente all'identità del parlante, poiché l'informazione è ormai incorporata nei token generati. Il meccanismo completo con cui l'embedding e i token del riferimento vengono combinati per realizzare la clonazione *zero-shot* è descritto nella Sezione 3.6.

3.6 Clonazione vocale

Descritti i singoli moduli, resta da illustrare come essi cooperino per realizzare la clonazione vocale *zero-shot*, ossia la riproduzione di una voce non vista in

[5]Nel codice il modulo corrisponde alla classe `Qwen3TTSSpeakerEncoder`. La variante CustomVoice specifica invece la voce tramite identificatori discreti (*speaker id*).

addestramento a partire da un breve campione di riferimento.

Il modello clona la voce fornendo una coppia di riferimento composta da audio e relativa trascrizione, secondo lo schema dell'In-Context Learning (ICL). L'audio di riferimento viene elaborato lungo due percorsi complementari: lo speaker encoder ne estrae l'embedding del parlante, che ne sintetizza l'identità vocale, mentre il tokenizer audio lo codifica nei corrispondenti codici RVQ. Questi elementi vengono combinati con la rappresentazione testuale del riferimento e del testo da sintetizzare per formare la sequenza di ingresso del backbone: l'embedding del parlante è collocato in una posizione fissa, mentre la coppia testo–audio di riferimento costituisce il contesto a partire dal quale il modello genera il nuovo enunciato. La voce di riferimento viene così trasferita al parlato sintetizzato senza alcun riaddestramento del modello.

Il condizionamento vocale effettivamente adottato nella pipeline implementata in questo lavoro è descritto nella Sezione 4.6.3.

La Figura 3.2 riepiloga il flusso completo della modalità di clonazione, dall'audio di riferimento fino alla forma d'onda finale.

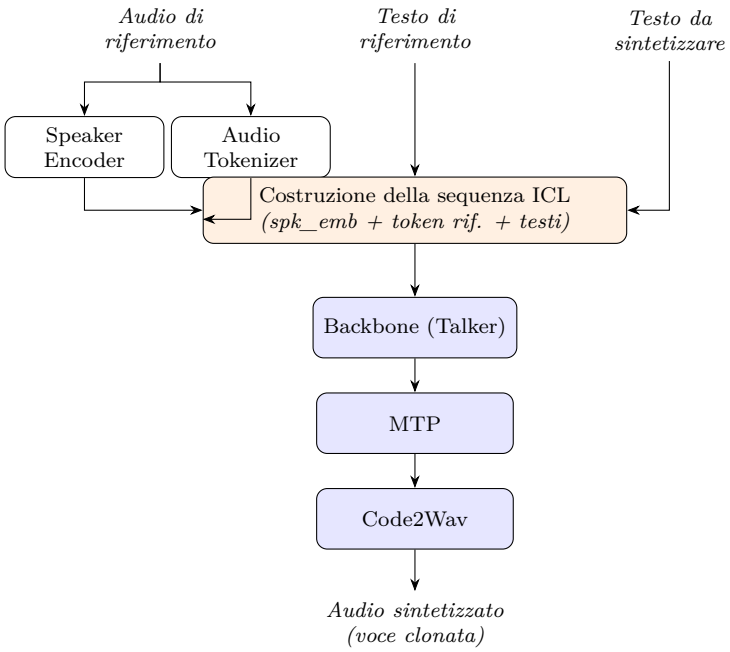


Figura 3.2: Flusso della clonazione vocale in modalità in-context learning (ICL). Dall'audio di riferimento lo speaker encoder estrae l'embedding del parlante e l'audio tokenizer i token RVQ; questi, insieme al testo di riferimento e al testo da sintetizzare, formano la sequenza di ingresso del backbone, che genera il parlato con la voce di riferimento.

Capitolo 4

Progettazione e implementazione della pipeline

4.1 Decomposizione della pipeline in componenti ONNX

Come descritto nel Capitolo 3, Qwen3-TTS è un sistema multi-stadio in cui la generazione di ciascun frame audio richiede l'esecuzione ripetuta e coordinata di più moduli, alcuni dei quali governati da cicli a lunghezza dinamica. Una simile struttura non è esportabile come un unico grafo computazionale ONNX: un grafo descrive infatti una sequenza statica di operazioni, mentre la generazione autoregressiva del modello, e in particolare il ciclo annidato del Code Predictor (Sezione 3.3), dipende da un flusso di controllo che si risolve solo a tempo di esecuzione.

La strategia adottata consiste pertanto nello **scomporre il modello in componenti indipendenti**, ciascuno esportato come grafo ONNX a sé stante, e nel ricostruire in codice C# la pipeline TTS: i cicli, la gestione della cache e il passaggio dei dati tra un componente e l'altro. In questo modo ogni grafo ONNX rappresenta una porzione puramente statica del calcolo, mentre la parte dinamica è affidata al codice applicativo che opera all'interno di Unity. La Figura 4.1 riepiloga i componenti in cui il modello viene scomposto e la loro derivazione dal modello PyTorch originale.

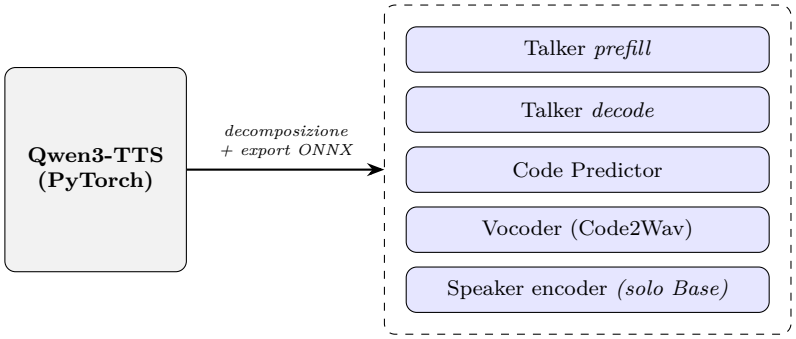


Figura 4.1: Scomposizione del modello in componenti ONNX indipendenti, gestiti dal codice applicativo.

La procedura di esportazione è stata messa a punto inizialmente sulla variante CustomVoice, che genera una voce predefinita e non richiede la gestione di un audio di riferimento, e successivamente estesa alla variante Base per abilitare la clonazione vocale. Poiché le due varianti condividono la medesima architettura, i componenti esportati sono in larga parte comuni: il Talker, il Code Predictor e il Vocoder risultano infatti **byte-identici** tra le due varianti, e l'unica differenza è costituita dallo speaker encoder, presente esclusivamente nella variante Base per l'estrazione dell'identità vocale dal riferimento. La Tabella 4.1 riepiloga i componenti e le rispettive dimensioni a valle dell'esportazione.

Tabella 4.1: Componenti ONNX della pipeline

Componente	**Ruolo**	**Dim. (FP32)**
Talker *prefill*	Inferenza iniziale sulla sequenza completa	1694,7 MB
Talker *decode*	Generazione autoregressiva del codebook 0	1694,7 MB
Code Predictor	Generazione dei codebook residui (1–15)	420,3 MB
Vocoder (Code2Wav)	Ricostruzione della forma d'onda	271,2 MB
Speaker encoder	Estrazione dell'embedding del parlante (solo Base)	34,0 MB

4.2 Metodologia di conversione Python-ONNX

L'esportazione di ciascun componente segue una procedura comune , che ricalca il flusso di esportazione standard di PyTorch [45] e l'impostazione della reference

implementation di ElBruno [46], adattandoli ai componenti di Qwen3-TTS. Questa sezione descrive nella sua forma generale. La procedura si articola in tre passi principali:

1. la definizione di un *wrapper* attorno al modulo PyTorch;
2. l'esportazione nel formato ONNX;
3. la validazione numerica dell'output.

I modelli così ottenuti sono stati poi sottoposti ad un'ulteriore trasformazione prima dell'impiego: la conversione in precisione ridotta (descritta nella Sezione 4.4).

Wrapper di esportazione. Ogni modulo di una rete neurale in PyTorch definisce il proprio calcolo nel metodo `forward`, che descrive come gli ingressi vengono trasformati nelle uscite quando il modello viene eseguito. I moduli di Qwen3-TTS non sono però direttamente esportabili nella loro forma originaria, poiché il loro metodo `forward` accetta e restituisce strutture dati complesse non rappresentabili in un grafo computazionale, che ammette unicamente tensori come input e output. Per ciascun componente è stato quindi definito un *wrapper*: un modulo che espone un'interfaccia costituita esclusivamente da tensori e che, al proprio interno, traduce tali tensori nelle strutture attese dal modulo originale, ne invoca il calcolo e ne riconverte le uscite.

Esportazione e validazione. Una volta definito il wrapper, il modello viene esportato utilizzando la funzione `torch.onnx.export` [45]. Affinché il grafo computazionale generato possa adattarsi a input di dimensioni variabili in fase di inferenza (come, ad esempio, sequenze testuali di lunghezza diversa), queste dimensioni vengono esplicitamente dichiarate come *assi dinamici*; in caso contrario, il modello accetterebbe solo input di una dimensione rigidamente prefissata.

L'output ottenuto viene infine sottoposto a validazione numerica per garantire che la conversione non abbia alterato la matematica del modello. Fornendo il medesimo ingresso di esempio sia al modulo PyTorch originale sia al grafo ONNX (eseguito tramite ONNX Runtime [27]), si calcola l'errore massimo in valore assoluto tra i tensori prodotti dalle due esecuzioni. Il codice 4.1 riporta la struttura ricorrente di questa procedura, esemplificata sul *prefill* del Talker.

```
# 1. Esportazione del wrapper in ONNX
torch.onnx.export(
    prefill_wrapper,
    (dummy_emb, dummy_mask, dummy_pos),      # ingressi di esempio
    out_path,
    opset_version=18,
```

```
    input_names=["inputs_embeds", "attention_mask", "position_ids"],
    output_names=["logits", "hidden_states", ...],  # + KV cache per layer
    dynamic_axes={"inputs_embeds": {0: "batch", 1: "seq_len"}, ...},
    dynamo=False,                                   # esportatore TorchScript
    do_constant_folding=True,
)

# 2. Validazione numerica PyTorch vs ONNX Runtime
sess = ort.InferenceSession(out_path, providers=["CPUExecutionProvider"])
ort_out = sess.run(None, {"inputs_embeds": dummy_emb.numpy(), ...})
diff = float(np.abs(logits_pt.numpy() - ort_out[0]).max())
assert diff < 1e-2      # tolleranza coerente con la conversione di precisione
```

Listing 4.1: Struttura della procedura di esportazione e validazione (Talker prefill).

Per quanto riguarda l'esito della validazione numerica, la soglia massima di errore accettabile è stata fissata a 10^{-2}, un valore coerente con l'errore atteso a seguito della conversione tra precisioni numeriche. Tuttavia, all'atto pratico, i componenti hanno mostrato una differenza massima tra PyTorch e ONNX Runtime tipicamente nell'ordine di 10^{-6}. Questo scarto quasi nullo rappresenta una conferma dell'estrema fedeltà della conversione rispetto al modello originale.

Gestione delle incompatibilità. Nonostante l'esportazione sia stata configurata per utilizzare l'*opset* 18[1], le versioni recenti di PyTorch e della libreria `transformers` introducono alcuni costrutti avanzati che l'esportatore ONNX non è ancora in grado di tracciare. Tra questi figurano le operazioni vettorializzate (`vmap`), il calcolo di differenze successive (`torch.diff`), le somme cumulate su tensori booleani e alcune specifiche implementazioni del meccanismo di attenzione.

Per risolvere sistematicamente queste incompatibilità, durante la fase di caricamento del modello è stato applicato un insieme di *patch*: si tratta di frammenti di codice che intercettano e sostituiscono a runtime i costrutti problematici con operazioni matematiche equivalenti, ma pienamente supportate da ONNX.

[1]Un *opset* (Operator Set) definisce la versione degli operatori matematici e neurali supportati dal formato ONNX. Versioni più recenti, come la 18, permettono a ONNX di tracciare e tradurre correttamente funzioni avanzate introdotte nelle ultime versioni di PyTorch.

Questo modulo di compatibilità è stato mutuato, senza modifiche, dal progetto `ElBruno.QwenTTS` di ElBruno [46][2].

4.3 Esportazione dei componenti

Esportato secondo la metodologia appena descritta, ciascun componente ha posto problemi specifici, legati alla presenza di costrutti che il formato ONNX o il runtime Sentis non sono in grado di gestire direttamente. Questa sezione esamina i componenti uno per uno, illustrando per ciascuno le difficoltà incontrate in fase di esportazione e le soluzioni adottate per superarle. Solo gli embedding, a differenza degli altri componenti, non vengono esportati come grafo ONNX ma estratti separatamente (Sezione 4.3.5).

4.3.1 Talker

Il Talker è stato esportato in **due grafi distinti** in corrispondenza delle due fasi della generazione autoregressiva. La fase di *prefill* elabora in un'unica passata l'intera sequenza di ingresso e ne inizializza la KV cache; la fase di *decode* genera quindi un token alla volta, riutilizzando a ogni passo la cache accumulata. Poiché i due regimi operano su forme dei tensori differenti, il prefill su una sequenza completa mentre il decode su un singolo token con cache in ingresso, la loro separazione in due grafi consente a ciascuno di lavorare con dimensioni ben definite, semplificando sia l'esportazione che l'esecuzione.

Questa suddivisione è comune a entrambe le varianti del modello: il Talker, come il Code Predictor e il Vocoder, è infatti byte-identico tra CustomVoice e Base (Sezione 4.1), e il wrapper di esportazione è il medesimo nei due casi. La Tabella 4.2 riassume le forme dei tensori coinvolti nelle due fasi.

Talker Prefill

La fase di prefill costituisce lo stadio preliminare dell'inferenza, in cui l'intero testo di input viene elaborato in un'unica passata dal modello. A tale scopo, il grafo riceve in input tre tensori fondamentali (Figura 4.2):

- `inputs_embeds`, di forma $(\text{batch}, T, 1024)$: gli embedding del testo. La dimensione *batch* è sempre pari a 1 (si elabora una frase alla volta), T è il numero di token del testo e 1024 la dimensione di ciascun embedding.

[2]Nello specifico, è stato utilizzato il file `compat_patches.py` dalla cartella `python/` del progetto.

- `attention_mask`, di forma (batch, T): indica quali token devono essere considerati. Qui tutti gli elementi valgono 1, poiché l'intera sequenza è costituita da token reali, senza padding da ignorare.
- `position_ids`, di forma $(3, \text{batch}, T)$: codifica la posizione di ciascun token nella sequenza, informazione necessaria al modello per tenerne conto dell'ordine. La prima dimensione, pari a 3, riflette l'uso di M-RoPE, che prevede tre assi posizionali.

Oltre a calcolare le rappresentazioni interne, l'elaborazione del prefill ha il compito di inizializzare le strutture dati per i passi successivi, producendo tre uscite:

- `logits_cb0`, di forma $(1, T, 3072)$: per ciascuna posizione, la distribuzione di probabilità (non normalizzata) sui 3072 possibili token del codebook semantico (in sostanza, quanto è probabile ciascun token audio come continuazione).
- l'`hidden_state`, di forma $(1, T, 1024)$: la rappresentazione interna di ciascun token, riutilizzata a valle nel ciclo di generazione.
- la KV cache: memorizza le rappresentazioni chiave-valore di ogni layer per riutilizzarle nella fase di decodifica senza il bisogno di ricalcolarle. Il grafo le restituisce come tensori separati (chiave-valore per ciascuno dei 28 layer) ciascuno di forma $[1, 8, T, 128]$: batch pari a 1, 8 teste chiave-valore (schema GQA, Sezione 3.2), T token elaborati e 128 come dimensione interna per testa. La loro ricomposizione in un unico blocco $[28, \ldots]$ è operata dalla pipeline prima di passarli al decode, non dal grafo stesso (Sezione 4.5).

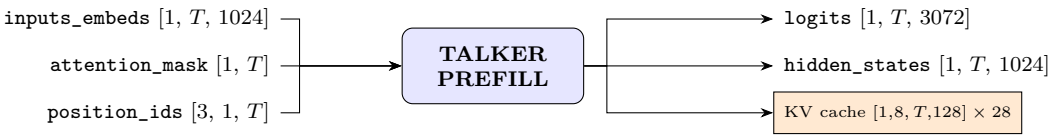


Figura 4.2: Input e output del Talker in fase di prefill.

Due aspetti, nella fase di esportazione, hanno richiesto un intervento specifico.

Il primo riguarda `position_ids`. Se non fornito esplicitamente, PyTorch lo calcola internamente durante il tracciamento e lo "congela" nel grafo ONNX come valore costante, legato alla lunghezza della sequenza di esempio usata in fase di export: il prefill risulterebbe corretto in isolamento, ma comprometterebbe la generazione successiva, che riceverebbe posizioni non aggiornate. Per questo motivo `position_ids` viene esposto come input a tutti gli effetti, nella forma a tre assi [3,1,T] richiesta dal M-RoPE (Sezione 3.2) anche quando, come si è visto, i tre valori coincidono numericamente.

Il secondo riguarda `codec_head`, lo strato lineare che proietta gli `hidden_states` sui logits del codebook 0. Includendolo direttamente nel wrapper, il grafo produce i logits in autonomia, senza che la proiezione debba essere ricalcolata a parte con una moltiplicazione di matrici esterna (Codice 4.2).

```
def forward(self, inputs_embeds, attention_mask, position_ids):
    outputs = self.model(
        input_ids=None,
        inputs_embeds=inputs_embeds,
        attention_mask=attention_mask,
        position_ids=position_ids,
        use_cache=True,
        return_dict=True,
    )
    hidden = outputs.last_hidden_state
    logits = self.codec_head(hidden)
    return logits, hidden, *kv_flat
```

Listing 4.2: Wrapper del Talker in fase di prefill (estratto).

Talker Decode

Una volta completato il prefill, la generazione dell'audio passa al grafo di decodifica, il quale opera in modo strettamente autoregressivo elaborando un singolo elemento alla volta. A ogni passo del ciclo, il grafo riceve quattro ingressi (Figura 4.3):

- `inputs_embeds`, di forma $(1, 1, 1024)$: l'embedding del token campionato al passo precedente (un solo token, non l'intera sequenza come nel prefill).
- la KV cache accumulata: gli stessi tensori chiave-valore prodotti finora, che la pipeline fornisce già ricomposti nella forma impilata $[28, 1, 8, T, 128]$ e che cresce di una posizione a ogni passo.
- `attention_mask` e `position_ids`, che si aggiornano a ogni passo per riflettere la posizione corrente nella sequenza.

L'esecuzione del passo di decode genera tre vettori di output fondamentali:

- `logits_cb0`, di forma $(1, 1, 3072)$: la distribuzione sul codebook semantico, da cui si campiona il token del frame successivo.
- l'`hidden_state`, di forma $(1, 1, 1024)$: la rappresentazione interna del frame appena elaborato, che alimenterà il Code Predictor.
- la KV cache aggiornata, $[28, 1, 8, T{+}1, 128]$, con il nuovo token accodato.

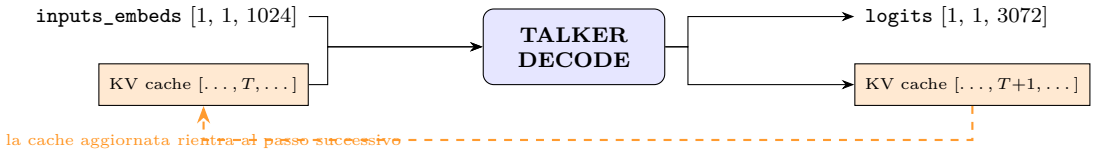


Figura 4.3: Input e output del Talker in fase di decode.

Anche la configurazione del wrapper di decodifica ha richiesto l'adozione di soluzioni specifiche per assecondare la natura iterativa del processo di export.

La prima riguarda `attention_mask`. La maschera deve crescere di una posizione a ogni passo di decodifica, così che ciascun nuovo token possa riferirsi a tutto ciò che lo precede; se restasse fissata alla lunghezza iniziale, il modello resterebbe cieco ai token via via generati.

La seconda riguarda il formato della KV cache. Tra un passo e il successivo, i tensori non sono esposti come elenco, bensì impilati in due soli blocchi di forma `[num_layers, 1, kv_heads, T, head_dim]`: una rappresentazione più compatta, adatta al passaggio ripetuto a ogni passo in Unity (Sezione 4.6). Internamente il wrapper ricostruisce da questi blocchi la struttura `DynamicCache`: l'oggetto con cui la libreria `transformers` gestisce la cache chiave-valore, discusso in dettaglio nella Sezione 4.3.2.

La tabella 4.2 riassume e mette a confronto le forme dei tensori nelle due fasi di prefill e decode.

Tabella 4.2: Forme dei tensori principali del Talker, prefill e decode a confronto.

Tensore	**Prefill**	**Decode**
`inputs_embeds`	$[1, T, 1024]$	$[1, 1, 1024]$
`attention_mask`	$[1, T]$	$[1, \text{past} + 1]$
`position_ids`	$[3, 1, T]$	$[3, 1, 1]$
`logits` (output)	$[1, T, 3072]$	$[1, 1, 3072]$
`hidden_states` (output)	$[1, T, 1024]$	$[1, 1, 1024]$

4.3.2 Code Predictor

Il Code Predictor genera i quindici codebook residui di ciascun frame attraverso un ciclo interno di quindici passi: a ogni passo produce un codebook, condizionandosi a quelli già generati. È quindi esso stesso un modello autoregressivo, con una propria cache chiave-valore.

A ogni iterazione, il Code Predictor riceve in ingresso tre tensori:

- `inputs_embeds`: le rappresentazioni vettoriali in ingresso. La sua forma e il suo contenuto variano tra il primo e i passi successivi:

  - al **primo passo** (generazione di cb_1), ha forma $(1, 2, 1024)$ ed è costituito dalla concatenazione dell'`hidden_state` del frame (prodotto a monte dal Talker Decode) e dall'embedding del codebook semantico cb_0 appena campionato.

  - ai **passi successivi** (generazione da cb_2 a cb_{15}), ha forma $(1, 1, 1024)$ ed è costituito unicamente dall'embedding del codebook predetto al passo immediatamente precedente.

- `step`: un tensore scalare che indica l'indice del codebook in fase di generazione (da 0 a 14).

- la KV cache, di forma $[5, 1, 8, T, 128]$ (5 come il numero di layer del Code Predictor): all'inizio di ogni frame è vuota ($T = 0$) e la sua lunghezza temporale T cresce di una posizione a ogni passo.

Al termine dell'elaborazione del passo, il grafo restituisce due tensori in uscita (Figura 4.4):

- I `logits`, di forma $(1, 1, 2048)$: descrivono la distribuzione di probabilità non normalizzata sui 2048 possibili valori del codebook corrente, da cui si campiona infine il token cb_i.

- La KV cache aggiornata, di forma $[5, 1, 8, T+1, 128]$: arricchita della nuova posizione e pronta per essere reimmessa come input al passo successivo.

Al termine dei quindici passi si dispone dei codici $cb_1, \ldots, cb_{15}$ che, uniti al codebook semantico cb_0, completano il frame.

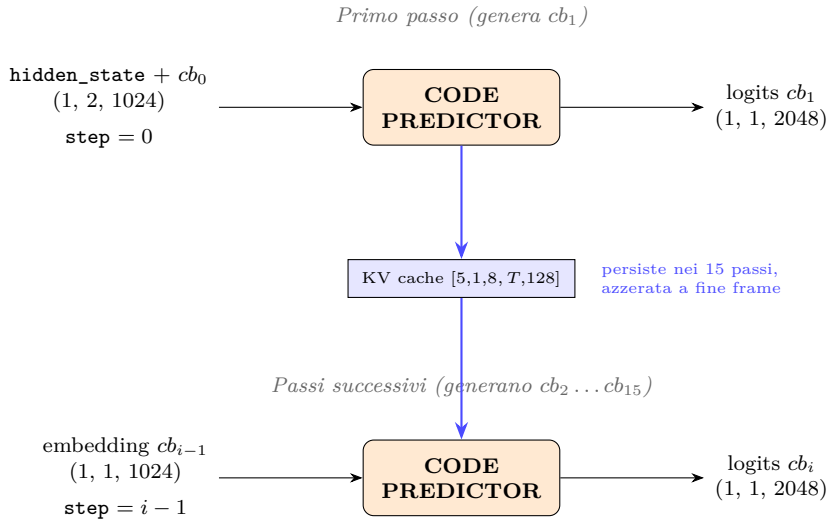


Figura 4.4: Input e output del Code Predictor

L'esportazione del Code Predictor ha posto alcune tra le sfide implementative più complesse dell'intero processo, richiedendo interventi mirati per aggirare i limiti del tracciamento statico di ONNX. Nello specifico, le criticità si sono concentrate su due fronti: la selezione dinamica delle teste di uscita e il corretto *tracing* della KV cache per la generazione autoregressiva.

Selezione dinamica delle teste di uscita

Come descritto nella Sezione 3.3, il Code Predictor impiega 15 teste di uscita distinte, una per ciascun codebook residuo. A ogni iterazione del ciclo interno di decodifica, il modello seleziona la testa appropriata in funzione del codebook attualmente in fase di generazione. Nell'implementazione nativa, queste teste sono organizzate all'interno di una `ModuleList` e indicizzate dinamicamente a ogni passo. Questo pattern di accesso a una lista di moduli non è tuttavia tracciabile dall'esportatore PyTorch: la costruzione di un grafo computazionale in ONNX richiede un flusso di calcolo statico e predeterminato, ed è incompatibile con la selezione condizionale di interi layer a tempo di esecuzione.

Per superare questa limitazione, le teste da moduli indipendenti sono state convertite in un'unica struttura dati. Anziché mantenere 15 livelli lineari separati, le rispettive matrici dei pesi sono state estratte e impilate tramite l'operazione `torch.stack` in un unico tensore tridimensionale, come illustrato in Figura 4.5.

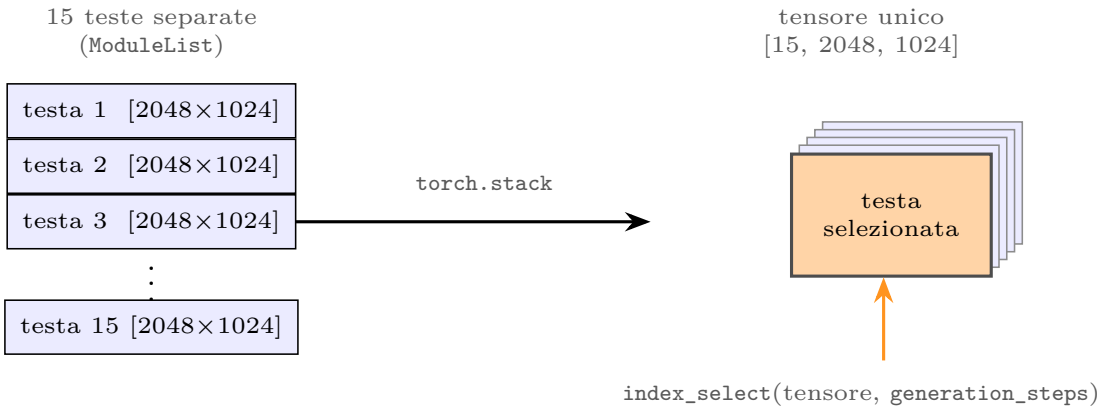


Figura 4.5: Stack delle teste di uscita.

In questo modo, l'individuazione della testa corretta viene ridotta all'estrazione di una specifica porzione di tale tensore attraverso l'operazione `index_select`. Essendo quest'ultima un'operazione tensoriale standard e nativamente supportata da ONNX, la dipendenza dinamica è stata eliminata preservando integralmente la matematica della rete. Concretamente, l'intera trasformazione si riduce a poche righe (Codice 4.3).

```
1 all_weights = torch.stack([h.weight for h in lm_heads])
2 self.register_buffer("lm_head_weights", all_weights)
3 ...
4 weight = torch.index_select(self.lm_head_weights, 0, step)
```

Listing 4.3: I pesi delle quindici teste sono impilati in un unico tensore; la testa corretta è selezionata con `index_select`, tracciabile in ONNX.

Tracciamento della cache autoregressiva

La seconda criticità ha riguardato la gestione della memoria temporanea (KV cache). Operando in modalità autoregressiva, il Code Predictor accumula progressivamente gli stati nascosti calcolati nei passi precedenti. L'implementazione originaria si affida all'oggetto `DynamicCache`, la cui logica interna prevede una biforcazione condizionale: se la cache è vuota alloca nuovo spazio, altrimenti concatena i nuovi valori a quelli preesistenti.

In fase di *tracing*, l'esportatore osserva ed "impara" esclusivamente il percorso logico innescato dall'input di prova fornito. Eseguendo l'esportazione a partire da una sequenza iniziale (con lunghezza temporale $T = 0$), l'esportatore registra unicamente il ramo di inizializzazione della cache. Di conseguenza, il grafo ONNX

risulta congelato su questa condizione e fallisce al secondo passo di inferenza, quando la cache richiede di essere aggiornata e non reinizializzata.

La strategia adottata per forzare la registrazione del grafo completo consiste nel tracciare l'esportazione fornendo in ingresso una cache pre-inizializzata (ad esempio, con $T = 2$). Grazie a questo accorgimento, l'esportatore percorre e memorizza il ramo di aggiornamento progressivo, che rappresenta il comportamento atteso e reiterato per tutta la fase di decodifica. La stabilità del modello esportato è stata infine comprovata validando l'output concatenato di due iterazioni successive, verificando la corretta propagazione degli stati interni rispetto alla controparte PyTorch.

4.3.3 Vocoder

Il Vocoder, ossia il decoder Code2Wav descritto nella Sezione 3.4, riceve in ingresso l'intera sequenza di codici dell'audio e ne ricostruisce la forma d'onda in un'unica passata.

Riceve un solo tensore:

- `audio_codes`, di forma $(1, 16, T)$: i codici quantizzati dell'intero enunciato, dove 16 è il numero di codebook per frame (il codebook semantico più i quindici residui) e T il numero di frame prodotti dalla generazione.

e produce un solo tensore (Figura 4.6):

- `waveform`, di forma $(1, 1, N)$: la forma d'onda PCM a 24 kHz, dove N è il numero di campioni audio risultanti.

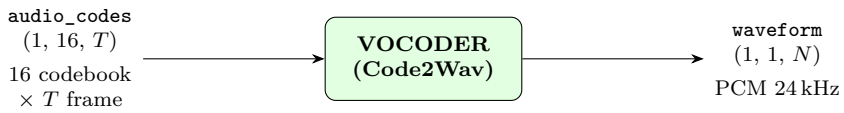


Figura 4.6: Input e output del Vocoder

L'esportazione di questo componente ha richiesto due interventi distinti in fase di esportazione: la decomposizione dell'attivazione SnakeBeta e il ricorso a un esportatore diverso da quello impiegato per gli altri componenti, reso necessario dal transformer interno al decoder.

Decomposizione dell'attivazione SnakeBeta

Come anticipato nella Sezione 3.4, il decoder impiega l'attivazione SnakeBeta. Diversamente dalle attivazioni standard, per le quali esiste un operatore ONNX

corrispondente, SnakeBeta è specifica del modello e priva di un operatore dedicato: il tentativo di esportarla direttamente fallisce, poiché l'esportatore non trova alcun operatore `SnakeBeta` nel vocabolario dell'opset.

La soluzione consiste nel **decomporre la funzione nei suoi operatori elementari**. SnakeBeta è definita, per ciascun canale, dalla relazione

$$\text{SnakeBeta}(x) = x + \frac{1}{e^{\beta} + \epsilon} \sin^2(x\, e^{\alpha}) \,,$$

con α e β parametri appresi. Sebbene ONNX non disponga di un operatore per l'intera funzione, ciascuna operazione che la compone (esponenziale, seno, potenza, moltiplicazione, divisione e addizione) è nativamente rappresentabile. È stata quindi realizzata una reimplementazione, `SnakeBetaONNX`, che ne esprime il calcolo esclusivamente tramite queste operazioni, preservandone la semantica ma rendendola interamente tracciabile (Codice 4.4). L'impianto di questa soluzione riprende l'approccio del progetto di riferimento di qwen-labs-cs [46], adattandolo al decoder impiegato in questo lavoro.

```
class SnakeBetaONNX(nn.Module):
    def __init__(self, original):
        super().__init__()
        self.no_div_by_zero = 1e-9
        params = dict(original.named_parameters())
        self.alpha = nn.Parameter(params["alpha"].data.clone())
        self.beta  = nn.Parameter(params["beta"].data.clone())

    def forward(self, x):
        alpha = torch.exp(self.alpha.unsqueeze(0).unsqueeze(-1))
        beta  = torch.exp(self.beta.unsqueeze(0).unsqueeze(-1))
        return x + (1.0 / (beta + self.no_div_by_zero)) * torch.
   sin(x * alpha) ** 2
```

Listing 4.4: Reimplementazione di SnakeBeta tramite operatori elementari tracciabili in ONNX.

Prima dell'esportazione, tutte le istanze di SnakeBeta presenti nel decoder vengono sostituite con la versione `SnakeBetaONNX`, verificando che la sostituzione non ne alteri l'uscita (differenza inferiore a 10^{-3}).

Esportazione con torch.export

Il decoder include al suo interno un modulo transformer. Nel costruire la maschera causale del suo meccanismo di attenzione, questo modulo ricorre a un'operazione vettorializzata che l'esportatore impiegato per tutti gli altri componenti (Sezione 4.2) non sa tradurre e, quindi, l'esportazione del Vocoder fallisce.

Per il Code2Wav si è quindi ricorsi a un esportatore alternativo, basato su `torch.export`[3], capace di gestire quell'operazione e di tracciare correttamente il transformer interno. A differenza degli altri componenti, il Vocoder non ha richiesto la definizione di un wrapper dedicato: viene esportato direttamente, con la sola sostituzione dell'attivazione SnakeBeta descritta sopra.

Il grafo così ottenuto è stato validato numericamente, con uscita coincidente a quella del modello originale entro la tolleranza stabilita.

4.3.4 Speaker Encoder

Lo speaker encoder, la cui architettura basata su ECAPA-TDNN è descritta nella Sezione 3.5, estrae dall'audio di riferimento l'embedding del parlante (un vettore di dimensione 1024) impiegato per il condizionamento vocale. La sua esportazione ha posto un problema di natura diversa dai precedenti, legato al suo strato finale. Nonostante l'esportazione in ONNX avvenisse con successo, il modello risultante non era caricabile in Unity: **Sentis non supporta l'operatore `If`**, che rappresenta un ramo condizionale nel grafo computazionale.

L'origine del nodo `If` è stata individuata nell'ultimo strato dell'ECAPA-TDNN. Il layer di proiezione finale contiene un controllo condizionale sulla forma del tensore: se l'ultima dimensione è pari a uno, essa viene eliminata (*squeeze*); in caso contrario il tensore è lasciato invariato. Durante l'esportazione questo controllo viene tradotto in un nodo `If` nel grafo.

L'analisi del flusso dei dati rivela però che la condizione è **sempre verificata**: il modulo di *attentive statistics pooling* che precede la proiezione produce infatti un tensore di forma fissa $[\text{batch}, 1024, 1]$, per cui il ramo *squeeze* è l'unico effettivamente percorso a tempo di esecuzione.

Poiché il ramo condizionale è ridondante, la soluzione adottata interviene direttamente sul grafo ONNX esportato: il nodo `If` e il calcolo della sua condizione vengono rimossi e sostituiti con un'operazione Squeeze incondizionata, mantenendo inalterata la semantica del modello.

```
# Il ramo squeeze, sempre percorso, viene reso incondizionato
squeeze_node = helper.make_node(
    "Squeeze",
    inputs=[conv_out_name, axes_name],    # axes = [-1]
    outputs=[if_output_name],             # 'speaker_embedding'
    name="/encoder/Squeeze_if_fix",
)
```

Listing 4.5: Sostituzione del nodo `If` con uno `Squeeze` incondizionato nel grafo dello speaker encoder.

[3] Attivato dall'opzione `dynamo=True` in `torch.onnx.export`.

La correttezza dell'intervento è stata verificata confrontando numericamente l'uscita del grafo originale, di quello modificato e del modello PyTorch a parità di ingresso: le tre uscite coincidono entro la tolleranza stabilita, confermando che la rimozione del ramo condizionale non altera il risultato.

4.3.5 Embedding e text projection

A differenza degli altri componenti, i layer di embedding del Talker e del Code Predictor non vengono esportati in ONNX. Questa è una scelta deliberata, non un'omissione: il grafo ONNX del Talker riceve in ingresso `inputs_embeds` già calcolato (Sezione 4.3.1), non un identificativo di token da cui ricavarlo. Poiché gli embedding restano costanti dopo l'addestramento, è più efficiente caricarli una sola volta in memoria come tensori grezzi piuttosto che includerli in un grafo pensato per un calcolo ripetuto ad ogni inferenza.

I pesi vengono quindi estratti direttamente dallo `state_dict` del modello PyTorch, salvati come file `.npy` e successivamente convertiti in file binari privi di intestazione (`.bytes`), leggibili in Unity come semplici array di `float` in memoria contigua. La Tabella 4.3 elenca i tensori estratti.

Tabella 4.3: Tensori di embedding estratti dallo `state_dict`, con relativa forma.

Tensore	**Forma**	**Origine**
Text embedding	$[151936, 2048]$	`text_embedding.weight`
Codec embedding (Talker)	$[3072, 1024]$	`codec_embedding.weight`
Codec head	$[3072, 1024]$	`codec_head.weight`
Text projection, fc1 (peso + bias)	$[2048, 2048]$	`text_projection.linear_fc1`
Text projection, fc2 (peso + bias)	$[1024, 2048]$	`text_projection.linear_fc2`
Codec embedding (Code Predictor)	$[2048, 1024] \times 15$	`code_predictor.codec_embedding.{i}`

Tra questi, i tensori della text projection (pesi e bias dei layer fc1 e fc2) meritano una nota a parte: anziché includere questa proiezione in un grafo ONNX dedicato, si è scelto di salvare i pesi grezzi per ricostruire interamente questo calcolo in modo nativo. Le motivazioni di questa scelta e l'implementazione del calcolo sono discusse nella Sezione 4.6.2.

4.3.6 Sinossi delle incompatibilità e delle soluzioni

La Tabella 4.4 riepiloga i costrutti non supportati incontrati e le relative soluzioni: le righe con un riferimento di sezione rimandano agli interventi specifici appena descritti, mentre quelle indicate come «compat.» sono risolte dal modulo di compatibilità mutuato (Sezione 4.2).

Tabella 4.4: Sinossi dei costrutti non supportati dall'esportazione ONNX e delle relative soluzioni

Costrutto / operatore	**Soluzione**	**Origine**
Operatore `If` (ramo condizionale)	Sostituzione con `Squeeze` incondizionato	§4.3.4
`position_ids` M-RoPE 3D calcolato internamente	Esposizione come input esplicito del grafo	§4.3.1
Accesso dinamico a `ModuleList` (teste di uscita)	Stack dei pesi + `index_select`	§4.3.2
`DynamicCache` tracciato a $T = 0$	Tracciamento con cache preinizializzata	§4.3.2
Attivazione `SnakeBeta` (operatore assente)	Decomposizione in operatori elementari	§4.3.3
`ROPE_INIT_FUNCTIONS ["default"]` rimossa da `transformers`	Reintroduzione di un'implementazione compatibile	compat. / §4.3.3
`sdpa_mask`: tensori 0-dim per `q_length`	Conversione esplicita a tensore scalare	compat.
`torch.diff` non tracciabile	Sostituzione con differenza tra sotto-tensori	compat.
`cumsum` su tensori booleani	Conversione a intero prima dell'operazione	compat.
GQA in SDPA (rottura del tracing)	Disabilitazione di `use_gqa_in_sdpa`	compat.
Masking basato su `vmap` (transformers 4.57+)	Registrazione di una variante priva di `vmap`	compat.

4.4 Quantizzazione

Per ridurre l'occupazione di memoria, i pesi dei componenti vengono convertiti dalla precisione a 32 bit (FP32) a quella a 16 bit (FP16). La Tabella 4.5 riporta le dimensioni prima e dopo la conversione: la riduzione è più marcata per il Talker, che viene interamente convertito, mentre è parziale per il Code Predictor e assente per il Vocoder, esportati già in precisione mista (Sezione 4.3.3).

Tabella 4.5: Dimensioni dei componenti ONNX prima e dopo la quantizzazione FP16. Talker convertito interamente, Code Predictor parzialmente; il Vocoder, già in precisione mista, resta invariato.

Componente	**FP32**	**FP16**	**Riduzione**
Talker prefill	1694,7 MB	850,8 MB	−50%
Talker decode	1694,7 MB	850,9 MB	−50%
Code Predictor	420,3 MB	270,4 MB	−36%
Vocoder	271,2 MB	271,2 MB	—

La conversione non può però essere applicata a tutti gli operatori del grafo. Alcuni di essi non elaborano valori numerici veri e propri, ma *indici* o *dimensioni* di tensori. Per queste operazioni la precisione ridotta è inadatta: un indice rappresentato in FP16 potrebbe perdere l'esattezza necessaria a individuare l'elemento corretto. Gli operatori di questo tipo vengono quindi esclusi dalla conversione e mantenuti in FP32.

Questa esclusione, però, introduce un rischio di incompatibilità: alcune operazioni aritmetiche legate alla normalizzazione, se lasciate in FP32, si troverebbero a combinare valori a precisione diversa (FP32 e FP16) all'interno dello stesso calcolo, generando un errore a tempo di esecuzione. Per evitarlo, tutte le costanti rimaste in FP32 vengono uniformate a FP16 in una fase di post-elaborazione, che precede alcuni interventi di pulizia del grafo (rimozione di nodi ridondanti e riordino delle operazioni).

4.5 Validazione end-to-end: la pipeline Python di riferimento

Esportati e validati singolarmente i componenti, il passo successivo è consistito nel verificarne il funzionamento congiunto. A questo scopo è stata realizzata una pipeline in Python, basata su ONNX Runtime, che coordina i vari grafi riproducendo l'intera logica di generazione. Questa pipeline ha svolto un duplice ruolo: da un lato ha permesso di validare l'insieme dei componenti in un ambiente rapido e flessibile, prima del porting; dall'altro ha costituito il riferimento algoritmico rispetto al quale è stata verificata l'implementazione nativa in C# (Sezione 4.6).

Avendo già dettagliato le interfacce di ingresso e uscita di ciascun grafo nella Sezione 4.3, l'attenzione si sposta qui sulla pipeline vera e propria, che si articola in tre fasi logiche: l'inizializzazione, il ciclo autoregressivo e la sintesi vocale. La Figura 4.7 ne offre una visione d'insieme, illustrando il flusso completo dei dati attraverso i vari componenti.

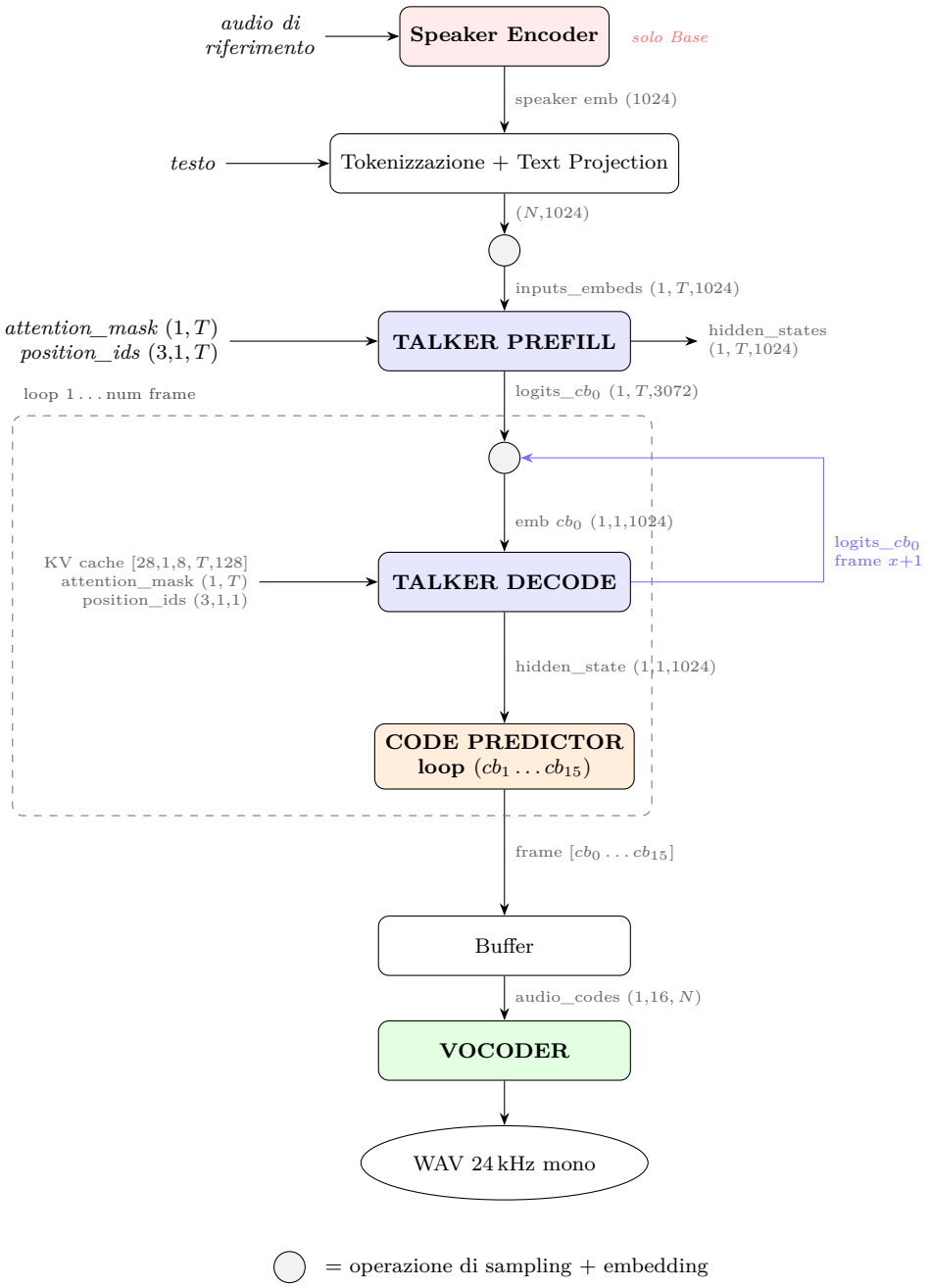


Figura 4.7: Flusso completo della pipeline di generazione. Il ramo dello *speaker encoder* è presente solo nella variante Base (clonazione vocale); nella variante CustomVoice la voce è definita da uno *speaker id* e tale ramo è assente

4.5.1 Inizializzazione e campionamento del primo token

Il processo prende avvio con la preparazione del contesto testuale e acustico. Il prompt in ingresso viene tokenizzato, proiettato in una sequenza di embedding e combinato con il condizionamento vocale.

Condizionamento vocale. Il condizionamento vocale differisce tra le due varianti. Nella variante Base, impiegata per la clonazione, la voce di riferimento è iniettata nella sequenza tramite lo *speaker encoder* (Sezione 4.3.4): l'audio di riferimento viene ricampionato a 24 kHz e convertito in feature mel (128 bande), da cui lo Speaker Encoder estrae l'embedding del parlante, un vettore di dimensione 1024. Questo embedding occupa una posizione fissa della sequenza di prefill e, condizionando l'intera generazione, fa sì che ogni token audio prodotto rifletta la voce di riferimento. Nella variante CustomVoice, invece, la voce è predefinita e indicata da un identificatore discreto (*speaker id*), senza ricorso allo speaker encoder.

Questa sequenza completa viene sottoposta al grafo di prefill del Talker in un'unica passata computazionale.

Campionamento del codebook semantico

La selezione del token relativo al codebook 0 non avviene in modo deterministico (scegliendo semplicemente il valore con probabilità massima), bensì attraverso una rigorosa procedura di campionamento stocastico (*sampling*). Questa scelta è fondamentale nelle architetture generative per garantire naturalezza ed espressività all'audio sintetizzato, evitando un output vocale monotono o artificiale.

Nel dettaglio, l'operazione di campionamento a partire dai logits dell'ultima posizione si articola in tre passaggi chiave:

1. **Filtraggio Top-K:** per circoscrivere la scelta ai soli candidati acusticamente plausibili, si scartano i valori marginali e si mantengono esclusivamente i 50 token con il punteggio logit più alto ($K = 50$).

2. **Normalizzazione Softmax:** i punteggi grezzi dei candidati trattenuti vengono convertiti in una vera e propria distribuzione di probabilità sommante a uno.

3. **Campionamento multinomiale:** il token definitivo viene estratto in modo pseudo-casuale, pesando la probabilità di estrazione di ciascun candidato in base alla distribuzione appena calcolata.

Al fine di evitare interruzioni premature dell'audio, nei primissimi passi della generazione la probabilità associata al simbolo di fine sequenza (EOS) viene forzata

artificialmente a zero. Il token così estratto (uno dei 3072 previsti dal vocabolario acustico) viene infine convertito nel corrispondente embedding vettoriale di dimensione 1024, fungendo da innesco effettivo per la successiva fase di *decode*.

Figura 4.8: Operazione di sampling ed embedding

4.5.2 Il ciclo di generazione a doppio livello

Ottenuto il primo token, la pipeline entra nel vivo della generazione operando attraverso due cicli annidati, illustrati nella Figura 4.9 e riportati nel Codice 4.6):

- **Il ciclo esterno (livello frame):** Governato dal grafo *decode* del Talker, gestisce l'avanzamento temporale dell'audio producendo un frame alla volta. A ogni step riceve l'embedding del frame precedente e aggiorna la KV cache estendendola di una posizione. Il suo compito è produrre due informazioni cruciali: l'`hidden_state` del frame corrente (che farà da matrice per il Code Predictor) e i nuovi logits da cui campionare il cb_0 del frame successivo. Questo ciclo prevede un limite massimo di salvaguardia di 1000 iterazioni (circa 80 secondi di audio), ma si interrompe autonomamente non appena estrae il token di fine sequenza.

- **Il ciclo interno (livello codebook):** Affidato al Code Predictor, si attiva per ogni singolo frame prodotto dal Talker. Sfruttando l'`hidden_state` appena generato, innesca un loop di 15 iterazioni esatte in cui produce, in sequenza, i codebook residui ($cb_1 \dots cb_{15}$). Poiché i codici all'interno di uno stesso frame sono temporalmente dipendenti, il Code Predictor accumula una propria KV cache, la quale viene però rigorosamente svuotata e reinizializzata all'inizio di ogni nuovo frame.

Al termine del ciclo interno, quando tutti i sedici codici del frame sono stati determinati, la pipeline ne estrae i rispettivi embedding, li somma tra loro e li combina con un vettore di contesto testuale (il cosiddetto *trailing*, che mantiene saldo il legame con la porzione di testo ancora da sintetizzare). Questo embedding consolidato del frame intero diventa l'ingresso per il passo successivo del ciclo esterno. La ricostruzione manuale della cache impilata [28, ...] tra un'iterazione e l'altra è proprio una delle operazioni di raccordo operate a livello Python in questa fase.

```
for step in range(MAX_TOKENS):
    token_cb0 = sample_group0(current_logits, step, prev_tokens)
    # top-K + multinomial
    if token_cb0 == CODEC_EOS:
        break
    codes = [token_cb0] + [0] * 15

    # Ciclo interno: Code Predictor genera cb1..cb15, con la
    propria KV cache azzerata
    cp_input = np.stack([hidden_state[0, -1], talker_codec(
    token_cb0)])[np.newaxis]
    cp_keys = cp_values = np.empty((CP_LAYERS, 1, CP_KV_HEADS, 0,
    CP_HEAD_DIM))

    for g in range(1, 16):
        out = sess_cp.run(None, {"input": cp_input, "step": [g-1],
                                 "past_keys": cp_keys, "
    past_values": cp_values})
        codes[g] = sample_cp(out[0][0, -1])
        cp_keys, cp_values = out[1], out[2]
        cp_input = cp_codec(g-1, codes[g])[np.newaxis, np.newaxis]

    # Ricombinazione del frame: somma degli embedding dei 16
    codebook + trailing
    next_emb = talker_codec(codes[0]) + sum(cp_codec(g-1, codes[g
    ]) for g in range(1, 16))

    # Talker decode: riusa la KV cache accumulata, produce i
    logits del frame successivo
    out = sess_decode.run(None, {"inputs_embeds": (next_emb +
    trailing[step])[None, None],
                                 "attention_mask": mask, "
    position_ids": pos_3d,
                                 "past_keys": past_keys, "
    past_values": past_values})
    current_logits, hidden_state, past_keys, past_values = out[0],
     out[1], out[2], out[3]
```

Listing 4.6: Struttura algoritmica del ciclo di generazione a doppio livello.

Il codice rende evidente come i singoli interventi descritti nelle sezioni precedenti convergano in un unico flusso funzionante: il ciclo interno sui codebook è l'esecuzione del meccanismo reso esportabile tramite lo *stack* delle teste di uscita; la reinizializzazione della cache del Code Predictor a ogni frame riflette la strategia di tracciamento con cache pre-inizializzata; e la costruzione del tensore `position_ids` tridimensionale attua la soluzione adottata per M-RoPE.

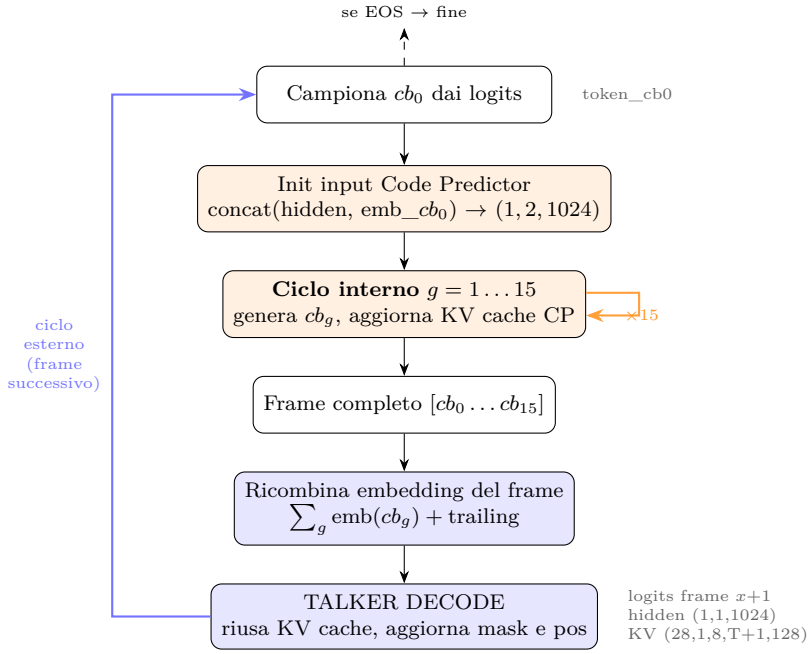


Figura 4.9: Dettaglio del ciclo di generazione a doppio livello

4.5.3 Sintesi in streaming

I frame interamente calcolati vengono accumulati in un *buffer* temporaneo. Al raggiungimento di una determinata soglia di frame (impostata per generare un "chunk" computazionalmente efficiente), il buffer viene svuotato e passato al Vocoder, che ne sintetizza la corrispondente porzione di forma d'onda a 24 kHz mono.

Questo approccio a blocchi consente all'audio di essere emesso in modo incrementale durante la generazione, anziché al termine dell'elaborazione dell'intera frase, abbattendo significativamente la latenza percepita dall'utente. Gli eventuali frame residui rimasti nel buffer vengono infine sintetizzati in blocco alla conclusione del ciclo esterno.

La correttezza matematica della pipeline è stata confermata dalla generazione di audio intelligibile e perfettamente coerente con la voce di riferimento, partendo unicamente dall'esecuzione dei componenti ONNX. Una volta validato questo prototipo, l'intera logica appena descritta è stata ritrascritta e ottimizzata per l'ambiente di destinazione finale, affrontato nella sezione seguente.

4.6 Porting in C#/Unity/Sentis

Il porting in ambiente Unity si fonda sul motore di inferenza Sentis, che esegue localmente i grafi ONNX prodotti nelle sezioni precedenti. A ciascun grafo è associato un *worker* Sentis distinto. Nella variante Base i worker sono cinque: oltre a quelli per il prefill e il decode del Talker, il Code Predictor e il Vocoder, uno è dedicato allo *Speaker Encoder*, che estrae dall'audio di riferimento l'embedding del parlante impiegato per il condizionamento vocale (nella variante CustomVoice, priva di Speaker Encoder, i worker sono quattro). Tutti sono configurati per l'esecuzione su GPU. I pesi di embedding e di text projection, estratti separatamente (Sezione 4.3.5), sono invece caricati in memoria come semplici array e impiegati direttamente nel codice applicativo.

Poiché la parte dinamica del calcolo non è esprimibile in un singolo grafo, l'intera logica di generazione autoregressiva, che nella pipeline Python di riferimento (Sezione 4.5) è gestita dall'interprete, viene qui ricostruita in C#. Ogni invocazione di un componente segue lo schema previsto dall'interfaccia di Sentis: si assegnano i tensori di ingresso al worker (`SetInput`), se ne pianifica l'esecuzione (`Schedule`) e se ne recuperano le uscite (`PeekOutput`). La struttura del calcolo ricalca quella della pipeline Python: un ciclo esterno sui frame audio, governato dal Talker in fase di decodifica, e un ciclo interno di quindici passi che, a ciascun frame, invoca ripetutamente il Code Predictor per generare i codebook residui.

L'aspetto più oneroso di questa ricostruzione riguarda la gestione manuale della KV cache, che in PyTorch è amministrata in modo trasparente dall'oggetto `DynamicCache` e che qui deve invece essere movimentata esplicitamente tra un'invocazione e la successiva. Al termine del prefill, i tensori della cache prodotti per ciascuno dei 28 layer vengono scaricati dalla memoria della GPU e reimpilati in due tensori complessivi, di forma `[28, 1, 8, T, 128]`, coerente con l'interfaccia attesa dal grafo di decode (Sezione 4.3.1). Ad ogni passo di decodifica successivo, questi tensori vengono forniti in ingresso e sostituiti con la loro versione aggiornata restituita in uscita, realizzando in codice applicativo la crescita incrementale della memoria del modello.

La cache del Code Predictor segue una logica distinta: essendo un modulo che rigenera i codebook residui in modo indipendente per ciascun frame, la sua cache viene inizializzata con lunghezza temporale nulla all'inizio di ogni frame e accresciuta nel corso dei quindici passi interni, per poi essere scartata al frame successivo. A valle del ciclo interno, l'embedding del frame completo viene ricomposto sommando il contributo del codebook semantico a quelli dei quindici codebook residui, e utilizzato come ingresso per il passo di decodifica seguente. Questa ricombinazione, come le proiezioni degli embedding, avviene interamente nel codice C# attorno ai grafi ONNX (Figura 4.10).

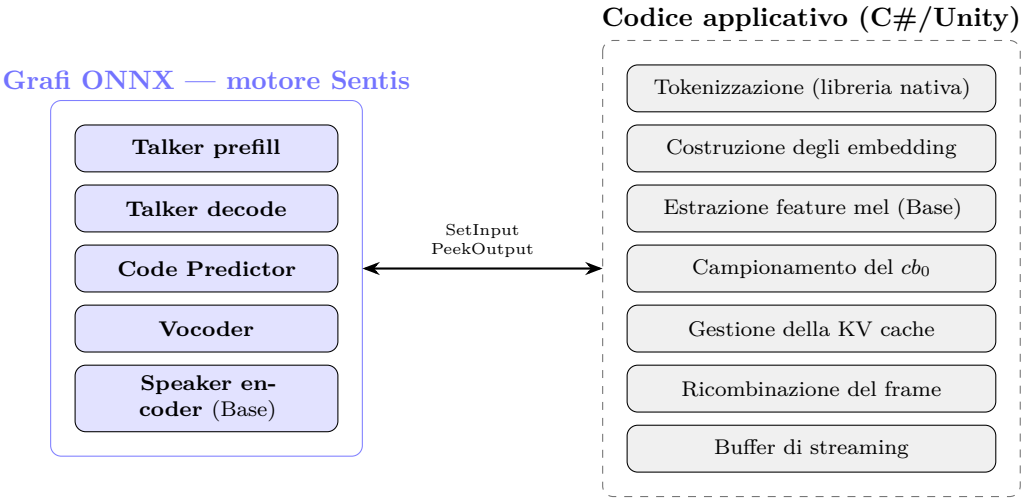


Figura 4.10: Divisione dei compiti nel porting in Unity

4.6.1 Tokenizzazione tramite libreria nativa

La conversione del testo in identificativi di token costituisce l'unico stadio della pipeline che non viene ricostruito in C#, ma delegato a una libreria esterna. Il tokenizzatore di Qwen3-TTS è basato sull'algoritmo Byte Pair Encoding (BPE)[4], il cui stato è interamente descritto da tre file: `vocab.json` per il vocabolario dei token, `merges.txt` per le regole di fusione e `config.json` per i parametri operativi. Reimplementare fedelmente la logica in C# sarebbe stato oneroso e fonte di potenziali discrepanze rispetto al comportamento originale. Si è preferito pertanto ricorrere a una libreria nativa che incapsula il tokenizzatore ufficiale, richiamata da Unity attraverso il meccanismo di interoperabilità con codice non gestito (P/Invoke).

La libreria, distribuita come DLL compilata a partire da un'implementazione in Rust[5], espone tre funzioni: una per l'inizializzazione a partire dai tre file di configurazione, una per la codifica di una stringa in identificativi di token e una per il rilascio delle risorse. La classe `QwenTokenizer` vi si interfaccia dichiarandone le firme (Codice 4.7) e traducendo le stringhe C# nella codifica attesa dalla libreria.

[4]Algoritmo di tokenizzazione a sottoparole (*subword*) che, partendo da un vocabolario di singoli caratteri o byte, fonde iterativamente le coppie di simboli più frequenti nel corpus di addestramento per creare nuovi token. Questo approccio permette di rappresentare parole comuni con un singolo token e parole rare o sconosciute assemblando token più piccoli, mantenendo il vocabolario di dimensioni contenute.

[5]Reperita dal bundle `Qwen3-TTS-ONNX-DLL` pubblicato su Hugging Face [47] e derivato dal tokenizzatore ufficiale di Qwen3-TTS; distribuita con licenza Apache-2.0.

```
1 [DllImport(DLL_NAME, CallingConvention = CallingConvention.Cdecl,
      CharSet = CharSet.Ansi)]
2 private static extern IntPtr qwen3_tts_tokenizer_create(string
      vocab, string merges, string cfg);
3
4 [DllImport(DLL_NAME, CallingConvention = CallingConvention.Cdecl)]
5 private static extern int qwen3_tts_tokenizer_encode(IntPtr handle
      , string text, long[] ids, int len);
6
7 [DllImport(DLL_NAME, CallingConvention = CallingConvention.Cdecl)]
8 private static extern void qwen3_tts_tokenizer_free(IntPtr handle)
      ;
```

Listing 4.7: Dichiarazione delle funzioni della libreria nativa di tokenizzazione (estratto).

All'avvio, i tre file di configurazione vengono individuati all'interno della cartella `StreamingAssets` e passati alla funzione di inizializzazione, che restituisce un riferimento opaco al tokenizzatore mantenuto in memoria per l'intera sessione. A ogni richiesta di sintesi, il testo di ingresso viene codificato in una sequenza di identificativi tramite la funzione dedicata; tali identificativi costituiscono l'ingresso dello stadio di costruzione degli embedding. Il riferimento al tokenizzatore viene infine rilasciato esplicitamente al termine del ciclo di vita del componente, per liberare le risorse allocate dalla libreria nativa.

4.6.2 Costruzione degli embedding fuori dall'ONNX

Come nella pipeline in Python, il grafo del Talker riceve in ingresso il tensore `inputs_embeds` già formato. La costruzione di questo tensore, che nella libreria PyTorch è svolta implicitamente dai layer di embedding del modello, è pertanto realizzata direttamente in C#, a partire dai pesi grezzi caricati in memoria. Due operazioni concorrono a questo scopo, illustrate nella Figura 4.11: la proiezione degli embedding testuali e la loro fusione con gli embedding del flusso codec.

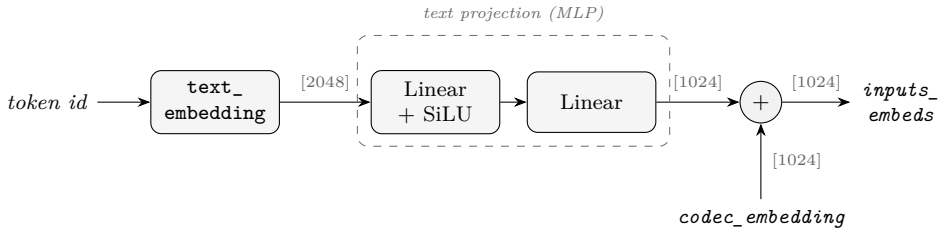


Figura 4.11: Costruzione dell'ingresso del Talker in codice applicativo: ogni token è convertito nell'embedding testuale, proiettato alla dimensione del backbone dalla *text projection* (un MLP a due strati con attivazione SiLU intermedia) e infine sommato all'embedding codec corrispondente.

Proiezione degli embedding testuali Ciascun identificativo di token prodotto dal tokenizzatore viene dapprima convertito nel corrispondente embedding testuale, di dimensione 2048, tramite un accesso diretto alla tabella `text_embedding`. Questo vettore viene poi proiettato alla dimensione 1024 del backbone attraverso la *text projection*, il cui calcolo è ricostruito esplicitamente: un primo strato lineare seguito dall'attivazione SiLU e un secondo strato lineare privo di attivazione, con i pesi dei due strati letti dai file estratti in fase di esportazione. L'operazione è raccolta in un metodo dedicato (Codice 4.8), invocato ogni volta che occorre proiettare uno o più token testuali.

```
Array.Copy(systemLoader.TextEmb, token * 2048, e, 0, 2048);
for (int j = 0; j < 2048; j++) {
    float sum = systemLoader.Fc1B[j];
    for (int k = 0; k < 2048; k++) sum += e[k] * systemLoader.Fc1W
    [j * 2048 + k];
    h[j] = sum / (1.0f + Mathf.Exp(-sum));      // SiLU
}
for (int j = 0; j < HiddenSize; j++) {
    float sum = systemLoader.Fc2B[j];
    for (int k = 0; k < 2048; k++) sum += h[k] * systemLoader.Fc2W
    [j * 2048 + k];
    output[j] = sum;                            // secondo strato,
    senza attivazione
}
```

Listing 4.8: Ricostruzione della text projection: accesso all'embedding testuale, primo strato con attivazione SiLU, secondo strato lineare (estratto).

Fusione con il flusso codec L'ingresso del Talker non è costituito dai soli embedding testuali, ma dalla loro combinazione con quelli del flusso codec, secondo il principio di fusione posizionale (Appendice A). Per ciascuna posizione della

sequenza, l'embedding testuale proiettato e il corrispondente embedding codec, ottenuto anch'esso per accesso diretto alla rispettiva tabella, vengono sommati elemento per elemento, producendo il vettore di dimensione 1024 che occupa quella posizione. La somma preserva la dimensione del vettore e realizza in codice applicativo esattamente la combinazione che il modello opererebbe internamente. Il tensore `inputs_embeds` così composto, comprendente il prefisso di ruolo, il prefisso codec e il corpo testuale della frase, costituisce l'ingresso del prefill del Talker.

Questa costruzione manuale degli embedding, unita alla gestione del flusso di generazione (Sezione 4.6), completa l'insieme delle operazioni eseguite in Unity attorno ai grafi ONNX: al motore di inferenza è lasciato solo il calcolo dei componenti neurali, mentre tutta la logica di preparazione dei dati, combinazione degli embedding e coordinamento dei passi di generazione risiede nel codice applicativo.

Attorno alla pipeline è stata infine sviluppata una semplice interfaccia di collaudo in Unity, che consente di selezionare la voce, immettere il testo e regolare i principali parametri di generazione:

- sintesi in streaming
- dimensione del chunk di streaming (frame accumulati prima dell'invio al Vocoder)
- numero di codebook residui da generare

Tale interfaccia (Figura 4.12) ha costituito lo strumento di prova per le valutazioni qualitative e per la raccolta dei dati di latenza discussi nella sezione successiva.

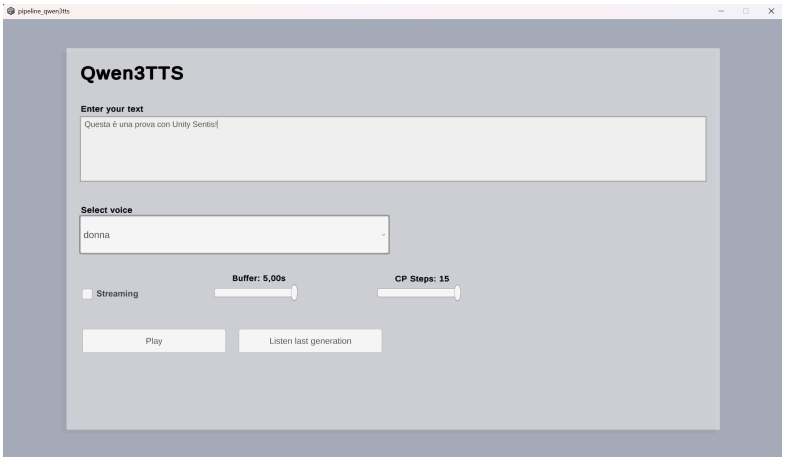


Figura 4.12: Interfaccia Unity

4.6.3 Condizionamento vocale nella variante Base

La variante Base realizza la clonazione vocale zero-shot condizionando la generazione su un embedding estratto dall'audio di riferimento, e richiede pertanto un componente assente nella variante CustomVoice: lo Speaker Encoder (Sezione 4.3.4), a cui è associato un quinto worker Sentis. La sua integrazione comporta un passo di pre-elaborazione da ricostruire in C#.

Lo Speaker Encoder riceve in ingresso non la forma d'onda grezza, bensì le sue feature Mel. L'audio di riferimento viene quindi convertito in uno spettrogramma Mel a 128 bande: la forma d'onda è suddivisa in finestre temporali brevi e trasformata nel dominio della frequenza tramite una *Short-Time Fourier Transform* (STFT), e le frequenze risultanti sono poi raggruppate in 128 bande percettive da un banco di filtri mel. Poiché Unity non dispone di una libreria di elaborazione del segnale equivalente a quelle impiegate in Python (dove questo passaggio è delegato a `librosa`), l'estrattore è stato reimplementato direttamente in C#.

Le feature Mel così ottenute sono assegnate al worker dello Speaker Encoder, che restituisce l'embedding del parlante, un vettore di dimensione 1024 che ne sintetizza l'identità vocale. Tale embedding viene infine inserito in una posizione fissa della sequenza di prefill, in luogo di un embedding codec, dove condiziona l'intera generazione. Poiché per una stessa voce l'embedding resta invariato, viene calcolato una sola volta e riutilizzato per tutte le sintesi successive con quel riferimento. Il condizionamento della variante Base è affidato al solo embedding del parlante; i codici RVQ e la trascrizione del riferimento previsti dalla modalità ICL (Sezione 3.6) non sono impiegati.

4.7 Analisi delle prestazioni

Oltre alla qualità del parlato, l'idoneità di una pipeline TTS a un impiego interattivo dipende dalla sua efficienza computazionale: in uno scenario in tempo reale la latenza è un requisito tanto importante quanto la naturalezza. Questa sezione ne analizza le prestazioni con tre obiettivi: individuare dove si concentra il costo di generazione, quantificare la reattività del sistema e valutare una possibile leva per ridurne la latenza.

A questo scopo è stata raccolta una serie di misurazioni durante la generazione, strumentando ciascuno stadio della pipeline in modo da registrare, per ogni frase sintetizzata, la latenza dei singoli componenti insieme alla durata dell'audio prodotto. Le misure sono state ottenute sulla variante CustomVoice in ambiente Unity, su un totale di 89 generazioni[6].

[6]Dal campione complessivo sono state escluse le generazioni "a freddo", in cui la prima

Le prove sono state eseguite su un portatile Lenovo LOQ (processore Intel Core i7, GPU NVIDIA GeForce RTX 4060 Laptop), con Sentis configurato per l'esecuzione su GPU.

L'analisi si articola in tre parti: l'individuazione del collo di bottiglia tramite la scomposizione della latenza per stadio; la caratterizzazione della reattività attraverso il RTF e la latenza al primo segmento audio; e, infine, lo studio di una tecnica per abbatterne il costo agendo sul componente dominante, ossia la riduzione dei codebook generati dal Code Predictor.

4.7.1 Individuazione del collo di bottiglia

La scomposizione della latenza per stadio (Figura 4.13) mostra con chiarezza dove si concentra il costo della generazione. Gli stadi iniziali (tokenizzazione, costruzione degli embedding e prefill del Talker) incidono in misura marginale, nell'ordine complessivo di poche centinaia di millisecondi; anche il Vocoder, pur ricostruendo l'intera forma d'onda, si mantiene su valori contenuti (circa 560 ms). La quasi totalità del tempo è invece assorbita dal **Code Predictor**, con una latenza mediana di circa 3,2 secondi per frase: da solo pesa più della somma di tutti gli altri stadi messi insieme.

La ragione risiede nella struttura del ciclo di generazione (Sezione 4.5): mentre il Talker produce un solo token per frame, il Code Predictor viene invocato **quindici volte** per ciascun frame, una per ogni codebook residuo. Queste quindici invocazioni sono per costruzione **sequenziali**: poiché la quantizzazione è di tipo residuo *coarse-to-fine*, ogni codebook si condiziona a quelli già campionati (Sezione 3.3) e non è quindi parallelizzabile. Nel runtime Sentis ciascuno di questi passi corrisponde inoltre a una distinta invocazione del worker (`Schedule`), con il relativo costo fisso di *dispatch*: è la ripetizione di tale overhead lungo il ciclo interno, più che il carico computazionale in sé, a rendere il componente dominante. All'interno del solo ciclo di decodifica, la fase di predizione multi-codebook rappresenta così da sola il **68 %** del tempo (mediana), contro il 32 % impiegato dal Talker decode e dalle operazioni di ricombinazione dell'embedding del frame. È dunque il Code Predictor a costituire il collo di bottiglia della pipeline.

---

invocazione di ciascun worker Sentis include il tempo di inizializzazione del modello; queste presentano una latenza di prefill fino a due ordini di grandezza superiore alla mediana e non sono rappresentative del funzionamento a regime.

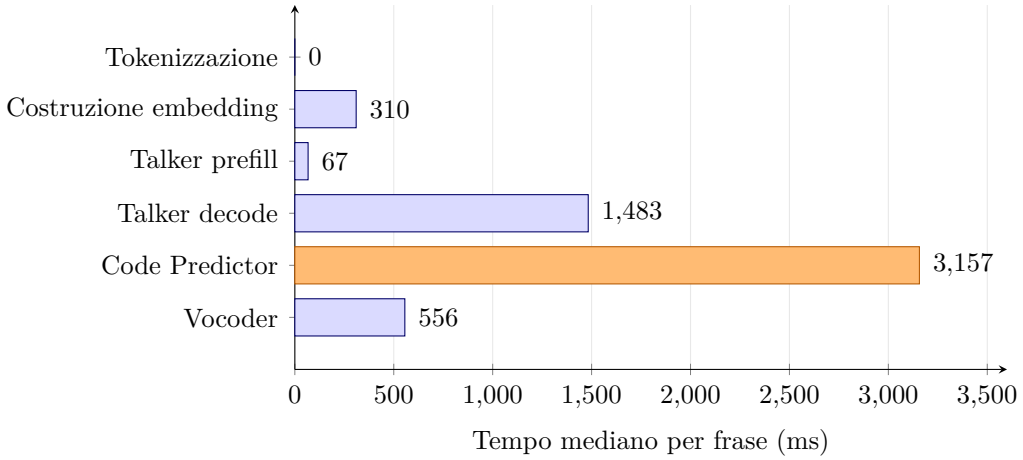
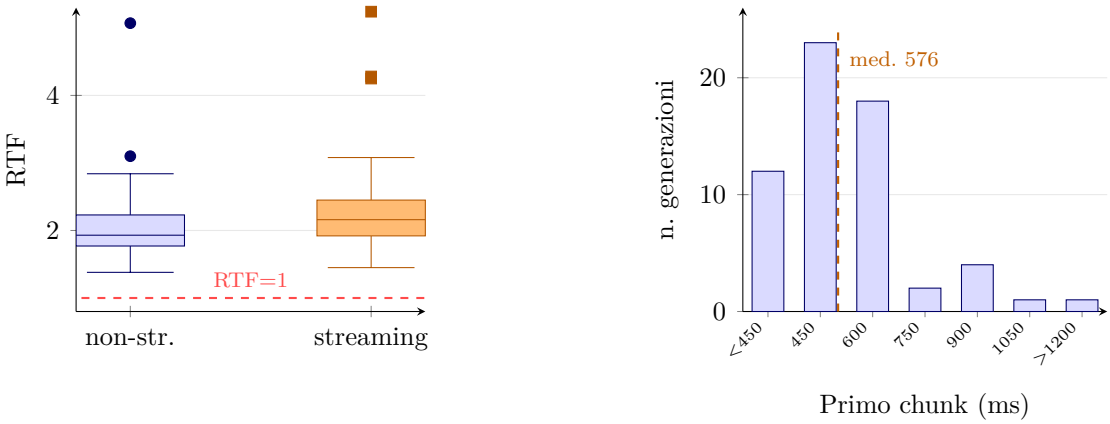


Figura 4.13: Scomposizione della latenza per stadio, in ordine di esecuzione (variante CustomVoice Unity)

4.7.2 Real-Time Factor e reattività

La Figura 4.14 riassume il comportamento temporale della pipeline nelle due modalità di generazione. In entrambe il RTF mediano si colloca sopra l'unità — circa 1,9 in modalità non-streaming e 2,2 in streaming — a indicare che l'audio viene prodotto a una velocità inferiore al tempo reale, coerentemente con l'overhead del ciclo multi-codebook individuato in precedenza. La modalità streaming, che introduce l'overhead della gestione a blocchi, presenta un RTF lievemente superiore; il suo vantaggio, però, non risiede nel tempo di generazione complessivo, bensì nella reattività percepita dall'utente.

È infatti la **latenza al primo segmento audio** (*time-to-first-chunk*) a determinare l'esperienza in uno scenario interattivo. In modalità streaming il primo blocco viene emesso dopo una mediana di circa **580 ms** (con un minimo di circa 350 ms), ben prima che l'intera sequenza sia stata generata: l'utente inizia così a sentire la risposta quasi subito, mentre il resto dell'audio viene prodotto durante la riproduzione. Sebbene la pipeline non raggiunga un RTF inferiore all'unità — condizione che garantirebbe uno streaming continuo senza interruzioni — questa reattività iniziale la rende adeguata a un impiego interattivo.



(a) Real-Time Factor

(b) Latenza al primo chunk (streaming)

Figura 4.14: A sinistra il Real-Time Factor nelle due modalità (linea tratteggiata: tempo reale); a destra la latenza al primo chunk in streaming.

4.7.3 Riduzione dei codebook residui

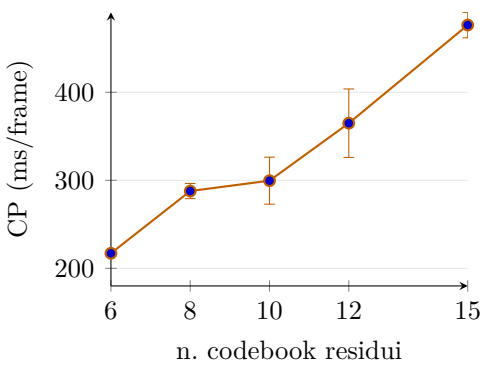
Individuato nel Code Predictor il collo di bottiglia, una leva diretta per ridurne il costo consiste nel diminuire il numero di codebook residui generati per ciascun frame. Ciò equivale a troncare la cascata di quantizzazione residua (RVQ): poiché i codebook seguono una gerarchia *coarse-to-fine*, in cui i primi livelli portano la struttura del segnale e gli ultimi soltanto il dettaglio acustico più fine, è possibile scartare i livelli più profondi conservando l'intelligibilità del parlato. In pratica il Code Predictor genera solo i primi N codebook residui (con $N < 15$), mentre i rimanenti vengono posti a zero prima di essere passati al Vocoder.

L'analisi è stata condotta sulla pipeline Python di riferimento (ONNX Runtime), sulla stessa macchina ma con esecuzione su CPU, sintetizzando ripetutamente lo stesso enunciato con $N \in \{6{,}8{,}10{,}12{,}15\}$ (otto ripetizioni per valore). I valori assoluti di latenza non sono quindi confrontabili con quelli misurati in Unity nelle sezioni precedenti; ciò che rileva è la *riduzione relativa*, valida in entrambi gli ambienti poiché il meccanismo, il Code Predictor invocato N volte per frame anziché quindici, è indipendente dal runtime.

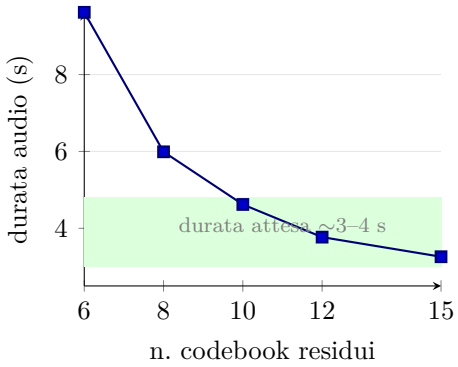
I risultati (Figura 4.15) evidenziano due effetti contrapposti. Da un lato, la latenza del Code Predictor per frame decresce in modo pressoché lineare con N (circa 32 ms per codebook), passando da 476 ms con 15 codebook a 217 ms con 6, con una riduzione del 37 % già a 10 codebook. Dall'altro, la qualità si degrada al calare di N: un indicatore oggettivo è l'allungamento incontrollato dell'audio generato a parità di testo: da circa 3,3 s con 15 codebook fino a 9,6 s con soli 6. Ciò rappresenta un segno della perdita di controllo del modello quando la rappresentazione acustica

si impoverisce. All'ascolto, la resa resta accettabile fino a circa 10 codebook, mentre al di sotto l'espressività emotiva si perde progressivamente e, con soli 6 codebook, la generazione diventa instabile.

Se ne ricava un compromesso: intorno a 10 codebook residui si ottiene una riduzione sensibile della latenza del componente dominante mantenendo una qualità ancora adeguata, mentre valori inferiori non risultano praticabili. La riduzione dei codebook costituisce dunque una leva efficace, ma vincolata da un netto trade-off qualità–latenza.



(a) Latenza del Code Predictor per frame



(b) Durata dell'audio generato (stesso testo)

Figura 4.15: Effetto della riduzione dei codebook residui

4.7.4 Sintesi e implicazioni

L'analisi delinea un quadro coerente. Il costo di generazione è fortemente concentrato: il Code Predictor, invocato quindici volte per frame, assorbe circa i due terzi del ciclo di decodifica e domina la latenza; ogni ottimizzazione ha senso solo se rivolta a questo componente.

Sul piano della reattività il quadro è però critico. Con un RTF mediano superiore all'unità (≈ 2), la pipeline genera l'audio più lentamente di quanto venga riprodotto: in uno streaming continuo, esaurito il buffer iniziale, la generazione non riesce a tenere il passo e si producono interruzioni della riproduzione (*buffer underrun*). La bassa latenza al primo segmento ($\sim$580 ms) migliora dunque soltanto l'avvio della risposta, ma non è sufficiente a garantire un flusso ininterrotto: **su un dispositivo di questa fascia la pipeline non soddisfa le condizioni per una sintesi in tempo reale propriamente detta**.

Va tuttavia sottolineato che questo limite non è di natura architetturale, ma dipende dall'implementazione e dal runtime: il modello è in grado di operare in tempo reale. Un'implementazione Python di riferimento per la generazione in

streaming di Qwen3-TTS raggiunge infatti un RTF inferiore all'unità, nell'ordine di 0,4–0,6 [48], agendo proprio sul ciclo interno del Code Predictor: la sua cattura come un unico grafo replicabile (CUDA graph, tramite torch.compile) elimina il costo di dispatch ripetuto a ogni passo, ossia esattamente la componente che domina nel port Sentis. Il divario di RTF osservato in questa tesi è dunque attribuibile al runtime di esecuzione. La via al tempo reale passa quindi, prima ancora che da hardware più potente, dalla riduzione dell'overhead per invocazione del ciclo interno affiancabile alla riduzione dei codebook analizzata sopra (efficace ma vincolata dal trade-off qualità–latenza) e alla distillazione del componente. Sul piano delle risorse, i pesi in precisione ridotta (Sezione 4.4) mantengono comunque l'occupazione di memoria entro i limiti della GPU impiegata, confermando la fattibilità dell'esecuzione on-device.

Capitolo 5

Valutazione soggettiva dell'espressività emotiva

I capitoli precedenti hanno descritto la costruzione della pipeline nativa e la sua validazione funzionale, dall'esportazione dei singoli componenti fino all'esecuzione in Unity. Resta tuttavia da affrontare un interrogativo fondamentale: in che misura il parlato sintetizzato riesce a trasmettere l'emozione desiderata? A differenza di proprietà come l'intelligibilità o la correttezza del contenuto, misurabili con metriche oggettive quali il *Word Error Rate*, l'espressività emotiva non dispone di un indice automatico affidabile: la sua resa dipende da sfumature di prosodia, ritmo e intonazione la cui adeguatezza è, in ultima analisi, giudicata dall'ascoltatore umano. Per questo la valutazione dell'espressività si affida tipicamente a test soggettivi, in cui sono le persone a esprimere una preferenza sul parlato ascoltato.

Come emerso nel Capitolo 2, per la lingua italiana nessun sistema open source coniuga la disponibilità dei pesi con un controllo emotivo esplicito: l'espressività è affidata prevalentemente al trasferimento dell'emozione tramite clonazione da un audio di riferimento. La valutazione presentata in questo capitolo misura pertanto proprio questa capacità, ossia quanto fedelmente ciascun modello riproduce la carica emotiva di riferimenti puramente espressivi, e la impiega per ordinare i sistemi analizzati e individuare quello più efficace sul piano dell'inflessione emotiva.

A questo scopo è stato condotto uno studio soggettivo di tipo crowdsourcing, in cui i partecipanti confrontano a coppie (A/B) clip audio pre-generate dai diversi modelli e scelgono quale rende meglio l'emozione indicata da un riferimento. I voti alimentano un sistema di ranking bayesiano basato sull'algoritmo TrueSkill [6], che stima l'abilità di ciascun modello come una distribuzione di probabilità e la aggiorna a ogni confronto; un meccanismo di campionamento adattivo seleziona

di volta in volta il confronto più informativo, riducendo il numero di valutazioni necessarie a ordinare i modelli in modo affidabile. Il capitolo descrive il disegno sperimentale e i modelli posti a confronto, il modello di ranking e la strategia di campionamento adattivo, l'applicazione sviluppata per la raccolta dei voti e, infine, i dati raccolti e i risultati ottenuti.

5.1 Design sperimentale

La valutazione mette a confronto i quattro sistemi TTS open source introdotti nella Sezione 2.6.2, tutti impiegati nella modalità di clonazione vocale a partire da un audio di riferimento: Chatterbox, OmniVoice, VoxCPM e Qwen3-TTS, quest'ultimo nella variante integrata nella pipeline nativa oggetto di questo lavoro. La scelta di limitare il confronto ai sistemi open source riflette l'obiettivo dello studio, ovvero valutare le soluzioni effettivamente eseguibili in locale e adattabili al dominio applicativo, escludendo i servizi commerciali a esecuzione remota.

Il confronto è condotto secondo un protocollo a **scelta forzata a coppie** (*two-alternative forced choice*): a ciascun partecipante viene presentata una coppia di clip audio, generate da due modelli diversi a parità di contenuto ed emozione, e gli si chiede di indicare quale delle due rende meglio l'emozione obiettivo. Rispetto all'assegnazione di un punteggio assoluto su scala, il confronto a coppie riduce il carico cognitivo del valutatore ed è meno soggetto ai bias di calibrazione della scala, che rendono difficilmente confrontabili i giudizi di persone diverse.

5.1.1 Stimoli

Gli stimoli sono clip audio **pre-generate**, prodotte una volta sola prima dell'avvio dello studio e poi riutilizzate in tutti i confronti. Ogni clip è definita da tre fattori:

- l'**emozione** obiettivo: le sei emozioni di base della tassonomia di Ekman [49] — rabbia, disgusto, paura, gioia, tristezza e sorpresa — più uno stato neutro di controllo, per un totale di sette categorie;
- la **lunghezza** della frase, su tre livelli (corta, media e lunga), così da osservare la resa emotiva sia su enunciati brevi sia su frasi più estese;
- il **modello** che l'ha generata, tra i quattro sistemi a confronto.

Per ogni combinazione di emozione e lunghezza sono state predisposte tre frasi distinte, così da non far dipendere il giudizio da un singolo enunciato. Ne risulta un totale di $7 \times 3 \times 3 = 63$ clip per modello, per complessive 252 clip. Tutte le frasi sono sintetizzate in modalità di clonazione a partire da un unico insieme di registrazioni di riferimento, una per emozione, comune a tutti i modelli: in questo

modo il confronto isola la capacità di ciascun sistema di trasferire l'emozione e non l'eventuale differenza tra i riferimenti.

5.1.2 Dataset di riferimento

Le registrazioni di riferimento che definiscono l'emozione obiettivo sono state costruite appositamente per questo studio e sono state generate con ElevenLabs [29] nella sua versione v3. La scelta è motivata proprio dal controllo emotivo esplicito di cui questo sistema dispone e che manca ai sistemi open source per la lingua italiana: sfruttando i marcatori espressivi che ElevenLabs consente di inserire nel testo (ad esempio `[furious, shouting]` per la rabbia o `[sobbing]` per la tristezza), è stato possibile ottenere per ciascuna emozione un riferimento dalla carica espressiva marcata e non ambigua. Ne risulta un dataset di sette clip, una per ognuna delle sette categorie, ciascuna basata su un breve enunciato in italiano scritto per evocare in modo netto l'emozione corrispondente.

Per ogni emozione la voce è stata selezionata individualmente tra quelle disponibili, privilegiando di volta in volta quella che ne restituiva la resa più convincente: nessuna singola voce si è infatti rivelata ugualmente efficace su tutte le emozioni. Di conseguenza le sette registrazioni impiegano voci differenti; ciò non pregiudica il confronto, che avviene sempre all'interno di uno stesso contesto (emozione e lunghezza fissate) e a parità di riferimento tra i modelli, ma privilegia deliberatamente la qualità espressiva di ciascun riferimento rispetto all'uniformità del parlante tra emozioni.

Questa impostazione riflette la logica dell'intera valutazione: si impiega un sistema commerciale, dotato di controllo emotivo esplicito, per fissare l'emozione bersaglio, e si misura quanto i sistemi open source, privi di tale controllo e costretti ad affidarsi alla sola clonazione, riescano a riprodurne l'inflessione. ElevenLabs non figura quindi tra i sistemi valutati, ma è impiegato unicamente per costruire i riferimenti: da un lato perché lo studio è mirato ai sistemi open source eseguibili in locale, dall'altro perché si tratta di un servizio commerciale a pagamento, il cui uso è stato circoscritto alla sola generazione una tantum del dataset di riferimento.

Tabella 5.1: Dataset di riferimento generato con ElevenLabs v3: per ciascuna emozione, la voce selezionata e il testo con i marcatori espressivi inseriti per guidarne la resa.

Emozione	**Voce**	**Testo con marcatori espressivi**
Rabbia	Carmelo La Rosa	`[furious, shouting]` BASTA! Non ne posso più! `[seething, voice cracking with rage]` Un'ora... UN'ORA per niente! `[explosive]` È UNO SCANDALO!
Disgusto	Marco	`[revolted, recoiling violently]` Che SCHIFO... `[retches]` non riesco neanche a guardarlo. `[nauseated, voice curling]` Mi viene da vomitare. Fa SCHIFO!
Paura	Danilo	`[gasps sharply]` `[terrified, hyperventilating]` C'è... c'è qualcuno in casa. `[whimpers]` `[voice shaking violently]` Oddio... non riesco a muovermi, ho troppa paura!
Gioia	Linda Fiore	`[laughs joyfully]` Che bello! Che bello tutto quanto! `[bright, breathless, ecstatic]` Mi viene da ridere e basta, non riesco a smettere. `[laughs, voice soaring]` Sto da Dio, sto proprio da Dio!
Tristezza	Alessandra	`[sobbing]` `[voice completely breaking]` Non ce la faccio più... `[wails softly]` mi manca tutto, da morire. `[voice dissolving into tears]` Non riesco a smettere di piangere.
Sorpresa	Nora	`[sharp gasp]` Cosa... `[stunned, frozen]` cosa sta succedendo?! `[breath caught, pitch jumping]` Ma... non me l'aspettavo. Cos'è stato?!
Neutro	Grandpa Spuds Oxley	`[calm, flat, detached]` Prendo tre medicine la mattina, una la sera. `[neutral, measured]` La pressione ieri era centoquaranta su novanta. `[tired, monotone]` Più o meno sempre così.

5.1.3 Compito del partecipante

Dopo aver espresso il consenso informato e fornito alcuni metadati anonimi facoltativi (fascia d'età, livello di conoscenza dell'italiano e genere), il partecipante svolge una sequenza di confronti. In ciascuno gli viene indicata l'emozione obiettivo e viene invitato ad ascoltare per intero due clip, denominate neutralmente A e B, per poi scegliere quale delle due trasmette meglio quell'emozione. Per garantire la validità del giudizio, il voto viene abilitato solo dopo l'ascolto completo di entrambe le clip. Ogni sessione propone dodici confronti; il partecipante può poi concludere o proseguire con un nuovo ciclo.

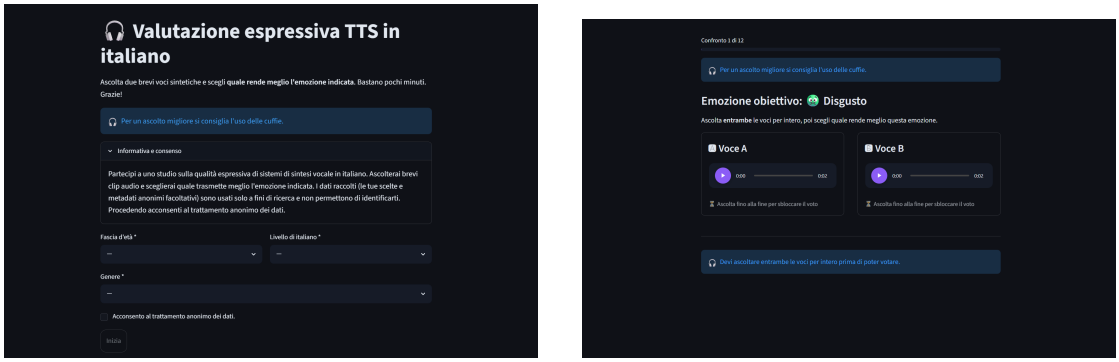


Figura 5.1: Interfaccia dell'applicazione di valutazione

L'assegnazione delle coppie non è casuale: un meccanismo di campionamento adattivo sceglie di volta in volta il confronto più informativo, in modo da concentrare i voti dove l'esito è ancora incerto e ridurre così il numero complessivo di valutazioni necessarie.

5.2 Il modello di ranking: TrueSkill

I voti raccolti nei confronti a coppie vengono aggregati in un ordinamento dei modelli tramite TrueSkill, un sistema di rating bayesiano sviluppato per stimare l'abilità di giocatori a partire dagli esiti di una serie di partite. L'idea di fondo è di non rappresentare l'abilità di un contendente con un punteggio fisso, ma con una **distribuzione di probabilità**, che viene aggiornata a ogni nuovo confronto e diventa via via più precisa con l'accumularsi delle osservazioni.

Originariamente concepito per il settore videoludico, TrueSkill è stato successivamente adattato con successo alla valutazione di sistemi complessi basata su confronti a coppie. Un riferimento consolidato in ambito accademico è la valutazione della traduzione automatica: l'algoritmo è divenuto il metodo standard per l'ordinamento delle preferenze umane nelle campagne di valutazione WMT (*Workshop on Machine Translation*) [50]. Il presente lavoro adotta tale approccio metodologico, estendendone l'applicazione a un dominio che, a differenza della traduzione testuale, rimane ancora privo di una metrica automatica universalmente riconosciuta: l'espressività emotiva del parlato in lingua italiana.

L'adozione di TrueSkill, in alternativa ai modelli di valutazione tradizionali, risponde a precise necessità del problema affrontato. A differenza del *Mean Opinion Score (MOS)*[1] [51], che richiede l'assegnazione di un punteggio assoluto su scala,

[1]Metrica soggettiva standard per la qualità del parlato, in cui ogni valutatore assegna a un singolo campione un punteggio assoluto su scala, tipicamente da 1 a 5.

il sistema basato su ranking per confronti neutralizza i bias di calibrazione tra valutatori diversi e garantisce un giudizio più semplice e robusto.

A un livello di implementazione tecnica, il vantaggio competitivo rispetto al sistema Elo[2] [52] risiede nella modellazione esplicita dell'incertezza σ associata ad ogni stima. Tale parametro permette di sapere in ogni momento quanto una stima sia consolidata, accelerando la convergenza dell'ordinamento con un numero ridotto di confronti. Soprattutto, la conoscenza di μ e σ consente di implementare il campionamento adattivo (Sezione 5.3), che seleziona i confronti più informativi per ottimizzare l'intero processo.

L'unità di cui si stima l'abilità è chiamata *player*. In questo studio un player corrisponde a una coppia **(modello, emozione)**: ciascuno dei quattro modelli possiede quindi una skill distinta per ogni emozione, per un totale di 28 player. In questo modo ogni emozione viene ordinata in modo indipendente dalle altre, coerentemente con l'osservazione che un modello efficace nel rendere la rabbia non lo è necessariamente nel rendere la tristezza. La lunghezza della frase non entra invece nella definizione del player: i confronti a lunghezze diverse concorrono alla stima della stessa skill, così da raccogliere più osservazioni per player senza moltiplicarne il numero.

Tabella 5.2: Composizione dei 28 player

Dimensione	**N.**	**Valori**	**Player?**
Modello	4	Chatterbox, OmniVoice, VoxCPM, Qwen3-TTS	Sì
Emozione	7	rabbia, disgusto, paura, gioia, tristezza, sorpresa, neutro	Sì
Lunghezza frase	3	corta, media, lunga	No
Totale: $4 \times 7 = 28$ player			

L'abilità di ogni player è modellata come una variabile aleatoria gaussiana $s \sim \mathcal{N}(\mu, \sigma^2)$, in cui:

- μ è la **stima corrente dell'abilità** del player (più alta, migliore è giudicato il modello per quell'emozione);
- σ è l'**incertezza** del sistema su tale stima: un valore alto indica che il player è stato ancora poco osservato.

[2]Sistema di rating ideato da Árpád Elo per la classificazione dei giocatori di scacchi, in cui l'abilità è riassunta da un unico punteggio numerico aggiornato dopo ogni partita.

La prestazione effettiva di un player in un singolo confronto non coincide con la sua abilità, ma oscilla attorno ad essa secondo $p \sim \mathcal{N}(s, \beta^2)$, dove β rappresenta la variabilità intrinseca di un singolo giudizio (il *rumore di prestazione*). Tutti i player sono inizializzati con gli stessi valori: $\mu_0 = 25$ e $\sigma_0 = 25/3 \approx 8{,}33$, con $\beta = 25/6 \approx 4{,}17$ e un fattore di dinamica $\tau = 25/300 \approx 0{,}08$, secondo i valori di riferimento tipici dell'algoritmo [6].

A ogni voto, che costituisce l'esito di un confronto uno-contro-uno tra due player, il sistema aggiorna per inferenza bayesiana le due distribuzioni coinvolte.

L'aggiornamento agisce su due fronti: **trasla** la media μ, che cresce per il vincitore e diminuisce per lo sconfitto, e **restringe** l'incertezza σ di entrambi, poiché il confronto ha aggiunto una nuova informazione. L'entità dello spostamento di μ è proporzionale alla "sorpresa" del risultato: un esito atteso, ovvero la vittoria del player con μ maggiore, comporta variazioni minime, mentre un esito inatteso corregge le medie in misura più marcata.

La sensibilità di μ dipende direttamente dall'incertezza corrente σ, cui è proporzionale. Con l'accumulo di votazioni, σ tende a ridursi e le stime diventano progressivamente più stabili. Se σ dovesse annullarsi, la media rimarrebbe di fatto congelata, rendendo impossibile qualsiasi ulteriore correzione anche a fronte di risultati sorprendenti. Per ovviare a questo limite, prima di ogni aggiornamento al valore σ viene sommato un fattore di dinamica τ. Quest'ultimo garantisce che l'incertezza si mantenga sempre sopra una soglia minima, preservando la reattività delle stime alle nuove osservazioni. In questo studio, in cui l'abilità dei modelli è considerata costante, τ non riflette un reale mutamento nelle performance, come avverrebbe tra le partite di un giocatore, ma funge da regolarizzatore, impedendo all'algoritmo di diventare eccessivamente sicuro delle proprie stime.

5.3 Campionamento adattivo

Il numero di confronti possibili cresce rapidamente con il numero di player, ma non tutti i confronti sono ugualmente utili: mettere ripetutamente a confronto due modelli il cui ordine è ormai stabilito, o di cui il sistema sa già molto, aggiunge poca informazione e spreca il tempo dei partecipanti. Per questo, anziché estrarre le coppie a caso, l'app adotta un **campionamento adattivo** che a ogni passo seleziona il confronto più informativo, così da ridurre il numero complessivo di voti necessari a ordinare i modelli in modo affidabile.

La selezione dei confronti più informativi non è un'esigenza specifica di questo studio, ma un problema ampiamente affrontato nella valutazione soggettiva della qualità, in particolare di immagini e video, dove ogni confronto è costoso in termini di tempo del valutatore. In quel contesto, algoritmi come ASAP [53] e Hybrid-MST [54] scelgono le coppie massimizzando il guadagno di informazione atteso. La

strategia adottata qui ne condivide il principio in una forma più semplice, fondata su un'euristica di urgenza anziché sul calcolo esplicito dell'informazione attesa.

Ogni confronto potenziale è individuato dalla tripla (emozione, lunghezza, coppia di modelli). Sebbene le diverse lunghezze della frase costituiscano confronti distinti, il sistema TrueSkill definisce il player esclusivamente sulla base della coppia (modello, emozione) (Sezione 5.2). Di conseguenza, i confronti che condividono emozione e coppia di modelli ma differiscono per lunghezza risultano tutti ugualmente urgenti, e la lunghezza viene di fatto selezionata a caso tra le tre disponibili. Individuata la lunghezza, la frase concretamente proposta viene sorteggiata tra le varianti di quella lunghezza, identica per entrambi i modelli così da garantire un confronto equo.

L'informatività di un confronto tra due player a e b è quantificata da un punteggio di **urgenza**, definito come:

$$\text{urgenza}(a, b) = \text{match_quality}(a, b) \cdot \sqrt{\sigma_a^2 + \sigma_b^2}, \tag{5.1}$$

dove il primo fattore misura la sovrapposizione tra le due distribuzioni di abilità e il secondo l'incertezza complessiva sui due player:

$$\text{match_quality}(a, b) = \sqrt{\frac{2\beta^2}{c^2}}\ \exp\left(-\frac{(\mu_a - \mu_b)^2}{2c^2}\right), \qquad c^2 = 2\beta^2 + \sigma_a^2 + \sigma_b^2. \tag{5.2}$$

L'urgenza è quindi alta quando l'esito del confronto è incerto (le due medie μ sono vicine, e le curve di abilità si sovrappongono) *e* quando i player coinvolti sono ancora poco conosciuti, ovvero hanno σ elevata. Viceversa, i confronti già di fatto decisi, in cui le due abilità sono nettamente separate, hanno match_quality ≈ 0 e dunque urgenza prossima a zero: vengono così scartati naturalmente, senza bisogno di soglie arbitrarie.

Ad ogni passo il sistema procede come segue: calcola l'urgenza per tutte le coppie candidate; con probabilità $\varepsilon = 0{,}10$ estrae una coppia a caso, così da preservare un minimo di esplorazione, mentre nel restante 90% dei casi propone la coppia a urgenza massima; presenta il confronto A/B, registra il voto e aggiorna di conseguenza le stime TrueSkill dei due player. Il ciclo si ripete, spostando via via l'attenzione sui confronti rimasti più incerti. (Appendice B)

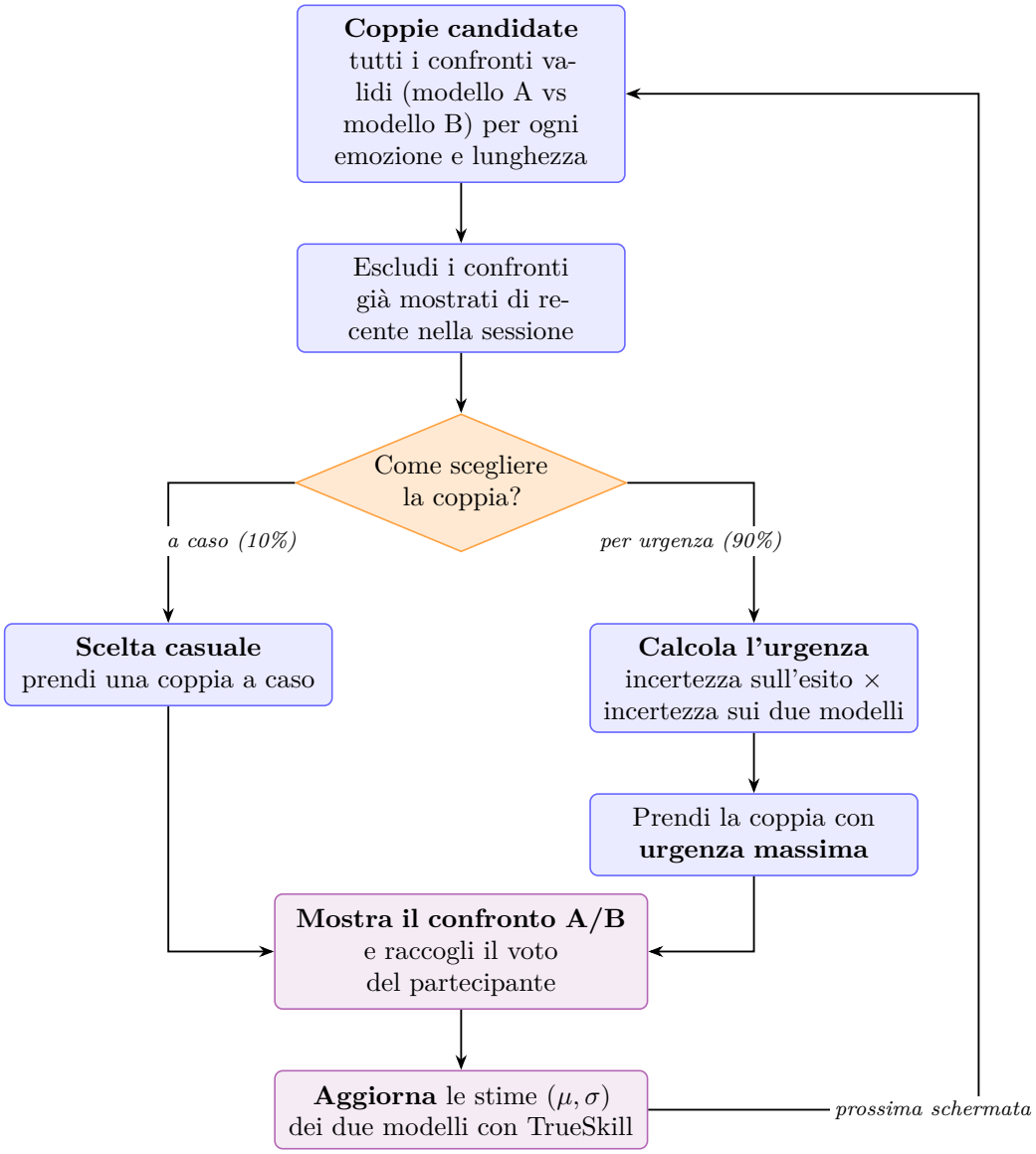


Figura 5.2: Ciclo di campionamento adattivo: a ogni schermata il sistema esclude i confronti recenti e, salvo una piccola quota di esplorazione casuale, propone la coppia a urgenza massima; dopo il voto aggiorna le stime TrueSkill dei due modelli.

5.4 Implementazione e raccolta dei dati

Per raccogliere i giudizi su un numero sufficiente di partecipanti, lo studio è stato condotto attraverso un'applicazione web sviluppata appositamente. Il frontend è realizzato con Streamlit e gestisce l'intero flusso di valutazione, mentre i dati

sono conservati in un database PostgreSQL ospitato su Supabase; l'applicazione è distribuita come servizio web raggiungibile tramite un semplice collegamento, così da poter essere condivisa con i partecipanti senza alcuna installazione locale.

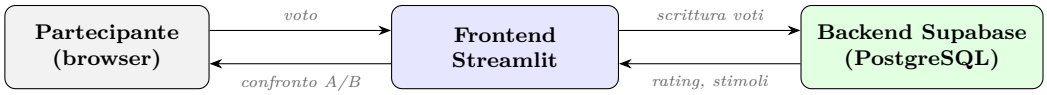


Figura 5.3: Architettura dell'applicazione di valutazione: il partecipante interagisce tramite browser con il frontend Streamlit, che gestisce i confronti A/B; i dati sono conservati nel database PostgreSQL ospitato su Supabase.

All'avvio, il partecipante prende visione di una breve informativa e presta il consenso al trattamento anonimo dei dati; facoltativamente fornisce alcuni metadati (fascia d'età, livello di conoscenza dell'italiano e genere), utili a caratterizzare il campione. Segue quindi una sessione di dodici confronti: per ciascuno gli viene indicata l'emozione obiettivo e vengono presentate due clip, denominate neutralmente A e B, con il voto abilitato solo dopo l'ascolto integrale di entrambe. (Figura 5.4) Al termine della sessione il partecipante può concludere o proseguire con un nuovo ciclo.

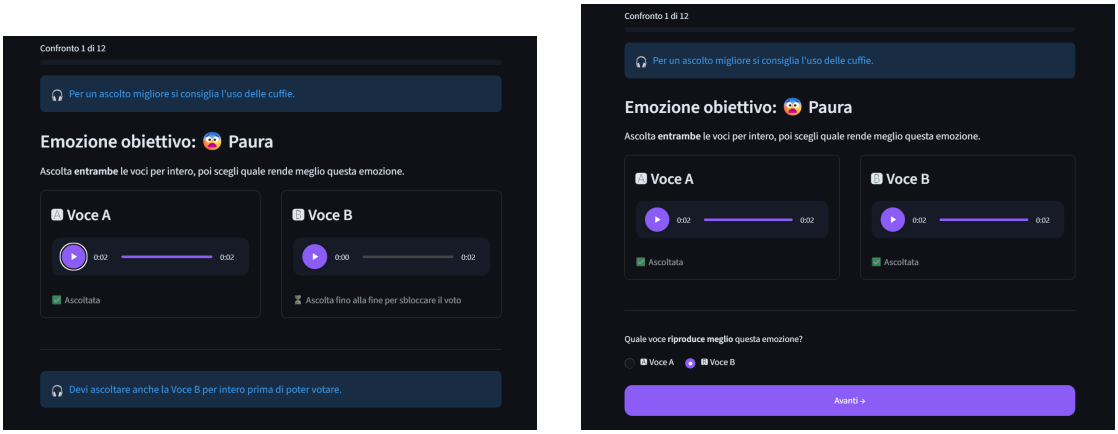


Figura 5.4: Schermata di confronto A/B

I dati raccolti sono organizzati in tre entità principali. La tabella `sessions` registra ogni partecipante con i relativi metadati anonimi e lo stato di avanzamento; la tabella `votes` conserva l'esito di ciascun confronto (emozione, lunghezza, i due modelli coinvolti, il vincitore, l'urgenza del confronto e l'istante di voto); la tabella `ratings` mantiene le stime TrueSkill correnti (μ, σ e numero di voti) di ogni player. A queste si affianca l'anagrafica dei modelli e degli stimoli audio.

Poiché più partecipanti possono votare simultaneamente, l'aggiornamento delle stime deve essere protetto dalle interferenze tra sessioni concorrenti. Ogni voto è pertanto elaborato da una procedura atomica lato database (`record_vote`), che in un'unica transazione, con blocco delle righe interessate, legge le stime dei due player, ne calcola l'aggiornamento TrueSkill e le riscrive, evitando così qualunque *race condition*. Per garantire coerenza, la matematica di TrueSkill è implementata in due punti allineati: una versione in Python, usata come riferimento e verificata numericamente, e una in PL/pgSQL interna al database, impiegata dalla procedura atomica; le due producono risultati coincidenti.

Sul piano etico, la partecipazione è anonima e subordinata al consenso informato; i dati raccolti, comprensivi dei soli metadati facoltativi, sono utilizzati esclusivamente a fini di ricerca e non consentono di identificare i partecipanti.

5.5 Risultati

In questa sezione vengono presentati i risultati della valutazione soggettiva, basati sull'analisi dei dati raccolti tramite la piattaforma di crowdsourcing.

5.5.1 Partecipanti e dati raccolti

Lo studio, condotto in modalità di crowdsourcing tramite la condivisione del collegamento all'applicazione, ha raccolto complessivamente **838 voti** distribuiti su **63 sessioni** di valutazione. Essendo ciascun voto l'esito di un confronto a coppie, esso concorre all'aggiornamento di due player, per un totale di 1676 aggiornamenti delle stime TrueSkill.

I partecipanti costituiscono un campione prevalentemente giovane e di madrelingua italiana, come riassunto nella Tabella 5.3. La netta prevalenza di madrelingua è particolarmente appropriata al compito, che richiede di cogliere sfumature prosodiche ed emotive del parlato italiano; la concentrazione nella fascia d'età 26–35 e lo sbilanciamento del campione sono invece tra i limiti discussi nella Sezione 5.7.

Tabella 5.3: Composizione del campione dei partecipanti ($N = 63$)

Età	**N**	**Livello**	**N**	**Genere**	**N**
18–25	9	Madrelingua	59	Maschio	38
26–35	47	Fluente	2	Femmina	24
36–50	2	Intermedio	1	N.D.	1
51–65	5	Base	1		

Grazie al campionamento adattivo, i voti si distribuiscono in modo sostanzialmente uniforme tra le sette emozioni (da un minimo di 99 per la sorpresa a un massimo di 149 per il disgusto) e tra le tre lunghezze di frase (286 confronti per le frasi corte, 282 per le medie e 270 per le lunghe), senza che alcuna condizione risulti sotto-rappresentata. Al termine della raccolta, il numero di voti per singolo player varia da circa 20 a 91, un valore sufficiente a stabilizzare le stime di abilità nella maggior parte dei casi.

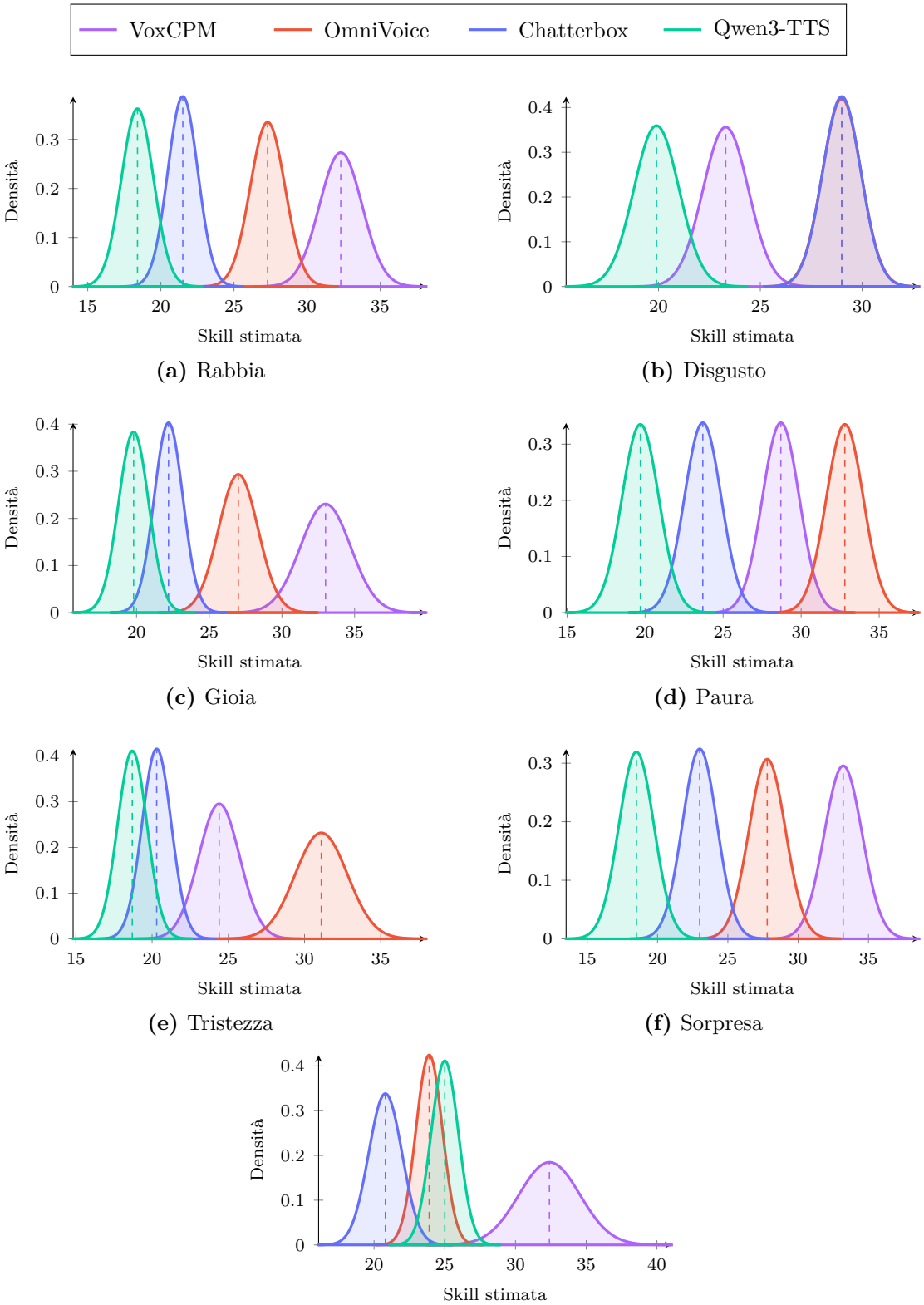
5.5.2 Analisi delle stime di abilità

Al termine della raccolta, le stime TrueSkill forniscono per ogni emozione un ordinamento dei quattro modelli in base alla media di abilità μ. La Tabella 5.4 riporta i valori ottenuti; per ciascuna emozione è evidenziato il modello risultato migliore. I valori completi di μ e σ per tutti i 28 player, insieme al numero di confronti su cui ciascuna stima si basa, sono riportati nell'Appendice C.

Tabella 5.4: Abilità stimata (μ) di ciascun modello per emozione e in media. Per ogni riga è evidenziato in grassetto il valore più alto. Valori più alti indicano una resa emotiva giudicata migliore dai partecipanti.

Emozione	**Chatterbox**	**OmniVoice**	**VoxCPM**	**Qwen3-TTS**
Rabbia	21,5	27,3	**32,3**	18,4
Disgusto	28,98	**29,0**	23,3	19,9
Paura	23,7	**32,8**	28,7	19,7
Gioia	22,2	27,0	**33,0**	19,8
Tristezza	20,3	**31,1**	24,4	18,7
Sorpresa	23,0	27,8	**33,2**	18,5
Neutro	20,8	23,9	**32,4**	25,0
Media	22,9	28,4	**29,6**	20,0

L'ordinamento complessivo vede al primo posto **VoxCPM**, seguito da vicino da **OmniVoice**; **Chatterbox** si colloca stabilmente terzo e **Qwen3-TTS** ultimo. Le prime due posizioni si alternano a seconda dell'emozione: VoxCPM prevale su rabbia, gioia, sorpresa e sullo stato neutro, mentre OmniVoice risulta il più efficace sulle emozioni negative più sfumate, ovvero disgusto, paura e tristezza. Il distacco contenuto tra i due, unito alla parziale sovrapposizione delle rispettive incertezze, suggerisce che nessuno dei due domini l'altro in modo netto su tutte le condizioni.



(a) Rabbia **(b)** Disgusto

(c) Gioia **(d)** Paura

(e) Tristezza **(f)** Sorpresa

(g) Neutro

Figura 5.5: Distribuzioni di abilità TrueSkill $\mathcal{N}(\mu, \sigma^2)$ dei quattro modelli per ciascuna emozione. La larghezza di ogni curva rappresenta l'incertezza della stima e la linea tratteggiata la media μ; la sovrapposizione tra due curve indica quanto i rispettivi modelli siano statisticamente distinguibili.

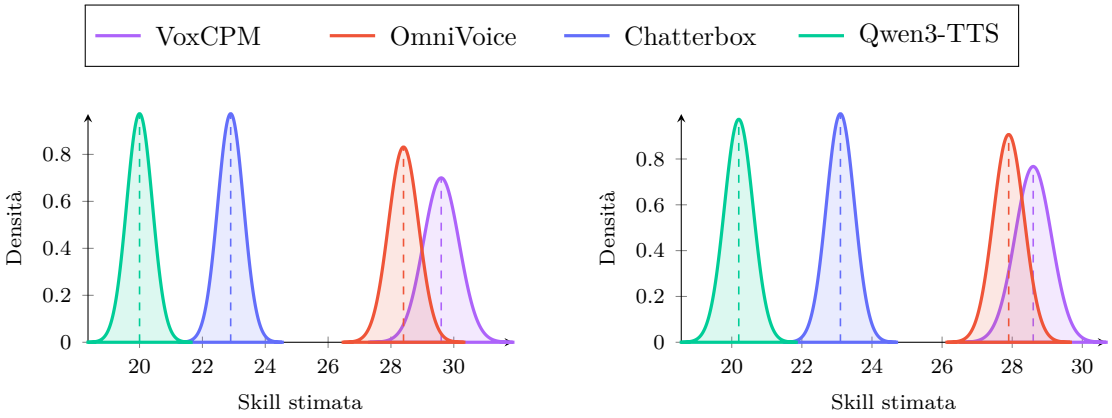
La pipeline sviluppata in questo lavoro, basata su Qwen3-TTS, ottiene i punteggi più bassi in sei emozioni su sette. Fa eccezione lo stato neutro, in cui risale al secondo posto: è il contesto in cui l'inflessione emotiva conta meno e in cui, di conseguenza, la resa più piatta del sistema viene penalizzata in misura minore. Va precisato il perimetro del confronto: a differenza degli altri sistemi, impiegati tramite le rispettive librerie originali, Qwen3-TTS è stato valutato nella variante integrata nella pipeline nativa, ossia il modello più compatto a 0,6 miliardi di parametri con condizionamento del parlante affidato al solo speaker embedding. L'esecuzione in Unity non incide su questo esito, poiché non altera l'audio generato ma solo le prestazioni (Sezione 4.7).

5.6 Aggregazione dei punteggi: tre scenari a confronto

L'analisi della Sezione 5.5.2 produce, per costruzione, sette ordinamenti distinti, uno per emozione, dato che ogni player rappresenta una coppia (modello, emozione). Per ricavare un **ordinamento complessivo** dei modelli occorre invece aggregare le sette skill di ciascun modello in un unico valore. Poiché non esiste un modo univoco di operare tale aggregazione, sono stati considerati tre scenari, che combinano gli stessi dati secondo ipotesi diverse; il loro confronto funge anche da verifica di robustezza dell'ordinamento.

- **Scenario I — media lineare.** Le sette distribuzioni di un modello sono mediate trattando ciascuna emozione allo stesso modo: $\mu = \frac{1}{n}\sum_i \mu_i$ e $\sigma^2 = \frac{1}{n^2}\sum_i \sigma_i^2$. È l'aggregazione più diretta, in cui ogni emozione pesa quanto le altre.

- **Scenario II — fusione (*sensor fusion*).** Le sette stime sono combinate come misure indipendenti di un'unica abilità sottostante, pesandole per la loro precisione: $\frac{1}{\sigma^2} = \sum_i \frac{1}{\sigma_i^2}$ e $\mu = \sigma^2 \sum_i \frac{\mu_i}{\sigma_i^2}$. Le emozioni stimate con maggiore certezza pesano di più, e l'incertezza complessiva risulta più bassa.



(a) Scenario I — media lineare **(b)** Scenario II — fusione per precisione

Figura 5.6: Abilità complessiva dei quattro modelli secondo i due criteri di aggregazione.

I risultati dei due scenari sono riportati nella Tabella 5.5.

Tabella 5.5: Ordinamento complessivo dei modelli secondo i due scenari di aggregazione (μ con, tra parentesi, l'incertezza σ). I due metodi concordano sull'ordine.

Modello	**Scenario I** *media*	**Scenario II** *fusione*
VoxCPM	29,6 (0,57)	28,6 (0,52)
OmniVoice	28,4 (0,48)	27,9 (0,44)
Chatterbox	22,9 (0,41)	23,1 (0,40)
Qwen3-TTS	20,0 (0,41)	20,2 (0,41)

Entrambi gli scenari producono il medesimo ordinamento — VoxCPM, OmniVoice, Chatterbox e infine Qwen3-TTS — a conferma che la classifica non dipende dal criterio di aggregazione scelto. I due criteri forniscono valori molto simili, con lo Scenario II leggermente più concentrato (incertezze inferiori), coerentemente con la sua natura di fusione pesata per precisione. La concordanza tra un'aggregazione a peso uniforme e una pesata per la precisione delle stime rafforza la validità della graduatoria complessiva.

5.7 Discussione e limiti

La valutazione raggiunge l'obiettivo per cui era stata concepita: ordinare i sistemi TTS open source in base alla resa dell'inflessione emotiva in italiano. L'esito è netto: gli ascoltatori hanno giudicato VoxCPM e OmniVoice i più espressivi: il primo avvantaggiato sulle emozioni ad alta attivazione e sullo stato neutro, il secondo sulle emozioni negative più sfumate. La pipeline basata su Qwen3-TTS è risultata la meno convincente in sei emozioni su sette. È una preferenza chiara e coerente tra i partecipanti.

Al di là del risultato, questa parte del lavoro ha un valore metodologico. L'espressività emotiva non dispone di una metrica oggettiva affidabile né di un benchmark consolidato per l'italiano, e misurarla richiede quindi un protocollo di giudizio umano. L'approccio adottato combina due pratiche consolidate: l'uso di TrueSkill per ordinare sistemi a partire da confronti a coppie, affermato nella valutazione della traduzione automatica [50], e il campionamento adattivo delle coppie, ampiamente impiegato nella valutazione soggettiva della qualità di immagini e video [53, 54]. Il contributo consiste nel trasferirli alla dimensione dell'espressività emotiva del parlato in italiano, ottenendo un ordinamento affidabile con un numero contenuto di giudizi e una stima quantificata dell'incertezza. Il protocollo resta applicabile al confronto di sistemi di sintesi su altre dimensioni percettive prive di metrica oggettiva.

Nell'interpretare l'ordinamento vanno considerati alcuni aspetti:

- **Natura del giudizio.** Il confronto a coppie misura una preferenza *relativa* tra sistemi a parità di contesto, non una qualità assoluta: indica quale voce rende meglio un'emozione, non quanto bene la renda in termini assoluti.
- **Riferimenti espressivi.** Ogni emozione è definita da un'unica registrazione di riferimento, con voci e generi differenti da un'emozione all'altra; la scelta, motivata dalla ricerca della resa più convincente per ciascuna emozione, comporta che l'identità del parlante non sia costante tra le emozioni. Il confronto avviene comunque sempre a parità di riferimento all'interno di una stessa emozione, il che ne circoscrive l'effetto.
- **Composizione del campione.** I partecipanti sono in larga maggioranza madrelingua italiani (requisito appropriato al compito) ma concentrati nella fascia d'età 26–35 e in numero limitato; le stime, robuste per l'ordinamento all'interno di ciascuna emozione, non sono necessariamente generalizzabili a popolazioni più ampie ed eterogenee.
- **Ambito del confronto.** Sono stati posti a confronto i soli sistemi open source; ElevenLabs è stato impiegato unicamente per costruire i riferimenti espressivi e non figura come termine di paragone valutato.

Capitolo 6

Conclusioni

Questo lavoro ha affrontato la sfida di implementare una pipeline di sintesi vocale espressiva che operi interamente in locale, progettata per essere integrata in sistemi di virtual humans. La ricerca è partita da un'esigenza concreta: per la lingua italiana non esisteva, allo stato attuale, una soluzione open source capace di coniugare qualità, espressività emotiva ed esecuzione nativa *on-device* all'interno di un motore di rendering. Questa necessità ha motivato lo sviluppo di una pipeline TTS integrata in Unity e la definizione di una procedura per la valutazione soggettiva della sua resa espressiva.

Dal punto di vista tecnico il modello Qwen3-TTS è stato scomposto, esportato in formato ONNX e adattato a Unity Sentis, risolvendo le criticità tecniche legate al porting da ambiente Python. Sul versante metodologico, è stato condotto uno studio di tipo *crowdsourcing* che, avvalendosi di confronti a coppie aggregati tramite TrueSkill e guidati da un campionamento adattivo, ha permesso di ordinare diversi sistemi open source in base alla capacità di trasmettere inflessioni emotive in lingua italiana.

L'analisi prestazionale ha confermato la fattibilità dell'esecuzione *on-device*: l'occupazione di memoria rientra nei limiti dell'hardware impiegato. Tuttavia, la pipeline non opera ancora in tempo reale: la generazione dell'audio risulta più lenta della riproduzione, confinando l'impiego operativo alla produzione asincrona rispetto all'interazione *live*. Il collo di bottiglia è stato identificato nel Code Predictor; tale limite non è di natura architetturale, ma imputabile all'overhead di invocazione del runtime Sentis durante il ciclo di decodifica dei codebook. Una futura ottimizzazione, finalizzata alla fusione del grafo di computazione, permetterebbe di abbattere tale latenza, avvicinando il sistema alle soglie del tempo reale.

Al di là delle attuali prestazioni temporali, il lavoro ha definito una metodologia di scomposizione e conversione (dal modello originale in PyTorch fino all'esecuzione dei grafi ONNX e alla gestione della memoria in C#) che costituisce un approccio

architetturale replicabile. Il percorso di integrazione tracciato fornisce infatti le linee guida per implementare all'interno del motore di rendering altri modelli acustici complessi, superando i limiti nativi dei runtime su dispositivo.

Sul fronte della valutazione, l'esito dello studio soggettivo è netto e concorde tra i partecipanti: VoxCPM risulta primo con entrambi i criteri di aggregazione, OmniVoice secondo, con le due posizioni che si alternano a seconda dell'emozione. VoxCPM prevale su rabbia, gioia, sorpresa e sullo stato neutro, OmniVoice sulle emozioni negative più sfumate (disgusto, paura, tristezza). Chatterbox si colloca stabilmente terzo e la pipeline basata su Qwen3-TTS ultima, risultando la meno convincente in sei emozioni su sette. Il confronto è circoscritto ai sistemi open source e Qwen3-TTS vi figura nella variante compatta integrata nella pipeline nativa, ma ciò non attenua un esito così marcato.

È proprio in questo esito che i due contributi, tecnico e metodologico, si completano. La pipeline è stata validata su Qwen3-TTS, un modello complesso da portare in Unity, confermando la solidità del percorso di integrazione; la valutazione, dal canto suo, ha fornito il criterio oggettivo per stabilire quale modello adottare. Che a risultare meno espressivo sia proprio il sistema integrato in questo lavoro dimostra, infatti, che il metodo di test non è stato costruito per confermare la scelta di partenza.

La procedura di valutazione qui definita, basata sull'adattamento di strumenti consolidati come il ranking bayesiano e il campionamento adattivo, ha dimostrato di essere una soluzione robusta per il benchmarking soggettivo di futuri sistemi di sintesi. Il suo perimetro attuale, per quanto statisticamente solido, resta circoscritto: un campione più ampio e diversificato, ed il potenziale affiancamento di metriche assolute ai confronti a coppie, ne estenderebbero ulteriormente la portata.

Autocertificazione dell'uso di AI generativa

Supporto editoriale

L'IA generativa ha svolto un ruolo editoriale sostanziale per la tesi, compresa la generazione di una bozza del documento.

Perfezionamento delle idee

Gli strumenti di IA generativa non sono stati utilizzati per il perfezionamento delle idee e il supporto editoriale.

Generazione di contenuti e supporto analitico

L'IA generativa ha fornito un contributo sostanziale con l'analisi dei dati, la generazione di codice, la formulazione di argomenti, la creazione di immagini/disegni o lo sviluppo di elementi di design. Indipendentemente dal livello di coinvolgimento dell'IA, tutti i contenuti generati dall'IA sono stati rivisti criticamente, verificati e modificati per allinearsi al rigore accademico. Le citazioni e le argomentazioni sono state verificate in modo indipendente. L'autore si assume la piena responsabilità dell'accuratezza, dell'originalità e dell'integrità del lavoro finale.

Appendice A

Verifica della fusione degli embedding nel Talker

Come anticipato nella Sezione 3.2, il technical report di Qwen3-TTS descrive l'approccio *dual-track* come una concatenazione dei flussi testuale e acustico lungo l'asse dei canali, mentre l'ispezione dell'implementazione rilasciata rivela che la fusione avviene tramite somma posizionale degli embedding. Questa appendice documenta la verifica condotta, articolata su tre livelli: l'analisi del codice sorgente, l'argomento dimensionale e una verifica numerica diretta.

Analisi del codice sorgente

Nel file `modeling_qwen3_tts.py` della libreria `qwen_tts`, tutte le operazioni di fusione tra il flusso testuale e quello acustico sono realizzate tramite l'operatore di somma (`+`), e non tramite `torch.cat`. Di seguito alcuni degli estratti rilevanti, tratti dal metodo `generate()`.

```
1  # riga 2019
2  text_embed = torch.cat(
3      [text_embed] + [tts_pad_embed] * (codec_lens - text_lens), dim
       =1
4  )
5  return text_embed + codec_embed, tts_pad_embed   # fusione (+)
6
7  # riga 2184
8  _talker_input_embed = torch.cat(
9      (tts_pad_embed.expand(-1, codec_input_embedding.shape[1] - 2,
       -1),
10      tts_bos_embed),
11     dim=1
12 ) + codec_input_embedding[:, :-1]                # fusione (+)
```

```
13
14 # riga 2201
15 talker_input_embed = torch.cat(
16     [talker_input_embed,
17      self.talker.text_projection(
18          self.talker.get_text_embeddings()(input_id[:, 3:4])
19      ) + codec_input_embedding[:, -1:]],           # fusione (+)
20     dim=1
21 )
```

Listing A.1: Fusione testo–codec tramite somma.

Le occorrenze di `torch.cat` presenti nel codice operano lungo l'asse temporale (`dim=1`), ossia concatenano token successivi nella sequenza; la combinazione tra il contributo testuale e quello acustico di una stessa posizione avviene invece sempre per somma.

Argomento dimensionale

La dimensione interna (*hidden size*) del backbone è pari a 1024. Come riportato nella Tabella A.1, la somma di due vettori da 1024 conserva la dimensione 1024, compatibile con il modello; una concatenazione ne produrrebbe invece uno da 2048, che richiederebbe un ulteriore livello di proiezione per essere riportato a 1024. Tale livello non è presente nel codice sorgente, il che rende la somma l'unica operazione consistente con l'architettura.

Tabella A.1: Confronto dimensionale tra le due possibili operazioni di fusione.

Approccio	**Operazione**	**Dim. output**
Somma (implementazione)	$\text{text_proj}_{[1024]} + \text{codec_emb}_{[1024]}$	1024
Concatenazione (report)	$\text{cat}(\text{text_proj}_{[1024]}, \text{codec_emb}_{[1024]})$	2048

Verifica numerica

Per confermare l'ipotesi si è confrontata la prima metà del vettore risultante dalla fusione con il solo flusso testuale. Nel caso di una concatenazione, la prima metà del vettore coinciderebbe esattamente con il flusso testuale; nel caso di una somma, i due contributi si sovrappongono canale per canale e le due quantità differiscono. I risultati confermano la somma:

- prima metà (somma) rispetto a `text_proj`: differenza massima = 0,103516 (valori distinti);

- prima metà (concatenazione) rispetto a `text_proj`: differenza massima = 0,000000 (identici per costruzione).

Poiché nell'implementazione reale la prima metà del vettore fuso non coincide con il flusso testuale, si conclude che i due flussi vengono effettivamente sommati.

Struttura della sequenza di prefill

A titolo illustrativo, la Tabella A.2 riporta la composizione delle prime posizioni della sequenza di ingresso del Talker in modalità streaming, evidenziando come, in ciascuna posizione, il contributo testuale e quello acustico si sommino per produrre un unico vettore di dimensione 1024. Si noti la posizione dedicata all'embedding del parlante (*speaker embedding*), impiegata nella modalità di clonazione vocale.

Tabella A.2: Composizione delle prime posizioni della sequenza di prefill (modalità streaming). Ogni posizione risulta dalla somma del contributo testuale e di quello acustico.

Pos.	**Contenuto**	**Flusso testuale**	**Flusso acustico**
0	*role* (im_start)	`text_proj`	—
1	*role* (assistant)	`text_proj`	—
2	*role* (newline)	`text_proj`	—
3	codec_think	`tts_pad`	`codec_emb`
4	codec_think_bos	`tts_pad`	`codec_emb`
5	lingua (es. italiano)	`tts_pad`	`codec_emb`
6	codec_think_eos	`tts_pad`	`codec_emb`
7	*speaker embedding*	`tts_pad`	`speaker_emb`
8	tts_bos + codec_pad	`text_proj`	`codec_emb`
9	text[0] + codec_bos	`text_proj`	`codec_emb`

Appendice B

Un esempio reale di campionamento adattivo

Qui viene illustrato, su dati reali raccolti durante lo studio, il funzionamento del campionamento adattivo introdotto nella Sezione 5.3. L'obiettivo è rendere concreto, su un caso specifico, il criterio con cui l'applicazione sceglie di volta in volta quale confronto proporre e l'effetto che ogni voto produce sulle stime. I valori di urgenza qui riportati coincidono con quelli effettivamente registrati nei log dell'applicazione al momento in cui ciascun confronto è stato proposto: la riproduzione dell'algoritmo a partire dai dati grezzi restituisce, per gli stessi stati, gli stessi punteggi, a conferma che l'implementazione realizza esattamente il criterio definito dall'Equazione 5.1.

L'urgenza di un confronto è il prodotto di due fattori: la *match_quality*, che misura quanto l'esito sia incerto (ed è massima quando le due distribuzioni di abilità si sovrappongono), e l'incertezza complessiva $\sqrt{\sigma_a^2 + \sigma_b^2}$, che è alta quando i due player sono ancora poco osservati. Un confronto è quindi tanto più informativo quanto più il suo esito è in bilico e quanto meno i modelli coinvolti sono già noti.

Un singolo passo. La Figura B.1 segue un passo del ciclo per l'emozione *sorpresa*, relativo alla coppia Chatterbox–Qwen3-TTS. Nello stato corrente i due player sono quasi indistinguibili: le stime di abilità valgono rispettivamente $\mu = 21{,}45$ ($\sigma = 3{,}01$) e $\mu = 20{,}16$ ($\sigma = 3{,}03$), e le due curve si sovrappongono ampiamente. Entrambi i fattori dell'urgenza sono dunque elevati: l'esito del confronto è incerto (medie vicine, forte sovrapposizione) e i due player sono ancora poco osservati (incertezze intorno a 3). Ne risulta un'urgenza pari a 3,40, la più alta tra tutte le coppie candidate in quel momento: è precisamente questo che porta il sistema a proporla.

Il partecipante ascolta le due clip e giudica Chatterbox più efficace nel rendere la sorpresa. Il voto viene incorporato tramite l'aggiornamento bayesiano di TrueSkill,

che agisce su due fronti: trasla le medie — quella del vincitore sale (da 21,45 a 22,30) e quella dello sconfitto scende (da 20,16 a 19,29) — e riduce entrambe le incertezze, poiché il confronto ha aggiunto informazione (σ passa da 3,01 a 2,85 e da 3,03 a 2,87). L'effetto congiunto sui due fattori dell'urgenza è una sua diminuzione: le medie sono ora più separate e le stime più consolidate, cosicché l'urgenza della coppia cala a 3,05. Al passo successivo, questa coppia risulterà quindi meno prioritaria e il sistema tenderà a spostare l'attenzione altrove.

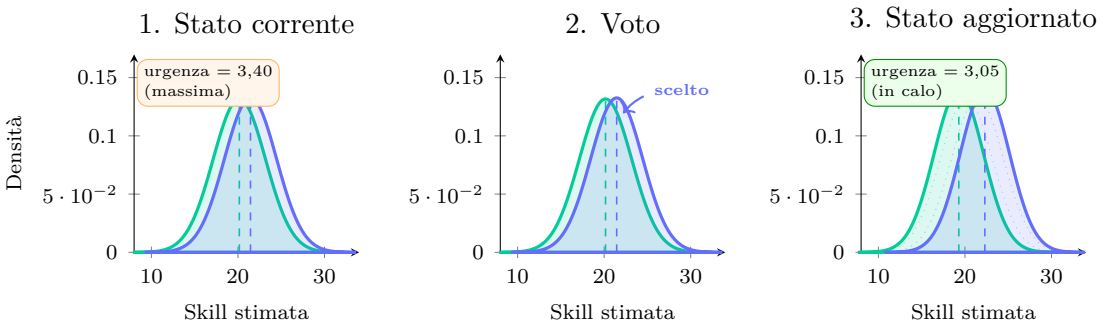


Figura B.1: Un passo di campionamento adattivo (emozione *sorpresa*, coppia Chatterbox–Qwen3-TTS): stato corrente, voto del partecipante e stato aggiornato. Le linee tratteggiate indicano le medie μ; in tratteggio puntinato, nello stato aggiornato, le distribuzioni precedenti.

L'effetto lungo più voti. Il comportamento appena descritto, ripetuto a ogni voto, produce nel tempo un progressivo consolidamento delle stime. La Figura B.2 lo mostra seguendo una singola coppia (VoxCPM–Chatterbox, sempre sull'emozione *sorpresa*) attraverso i suoi confronti successivi. Al primo confronto i due player si trovano ancora al valore iniziale ($\mu = 25$, $\sigma = 8{,}33$): l'incertezza è massima e le due distribuzioni coincidono, il che rende l'urgenza pari a 5,27, il valore più alto possibile. A ogni successiva comparsa della coppia le distribuzioni si restringono e le medie si separano, e l'urgenza decresce di conseguenza: $5{,}27 \to 4{,}85 \to 3{,}61 \to 2{,}56$.

Si noti che le medie non si spostano in modo perfettamente monotòno da un pannello all'altro: ciò accade perché, tra un'apparizione e la successiva, ciascun player affronta anche gli altri modelli sulla stessa emozione, e la sua stima riflette l'insieme di quei confronti. L'andamento sistematico, comune a tutte le coppie, è però la riduzione dell'incertezza e il consolidarsi dell'ordine relativo, che insieme fanno calare l'urgenza. Man mano che l'esito diventa più certo, la coppia viene riproposta sempre meno e i voti residui si concentrano sui confronti ancora in bilico. È questo il meccanismo per cui il campionamento adattivo raggiunge un ordinamento affidabile con un numero di valutazioni sensibilmente inferiore rispetto a un'estrazione casuale uniforme, in cui i confronti già decisi continuerebbero a essere proposti con la stessa frequenza di quelli incerti.

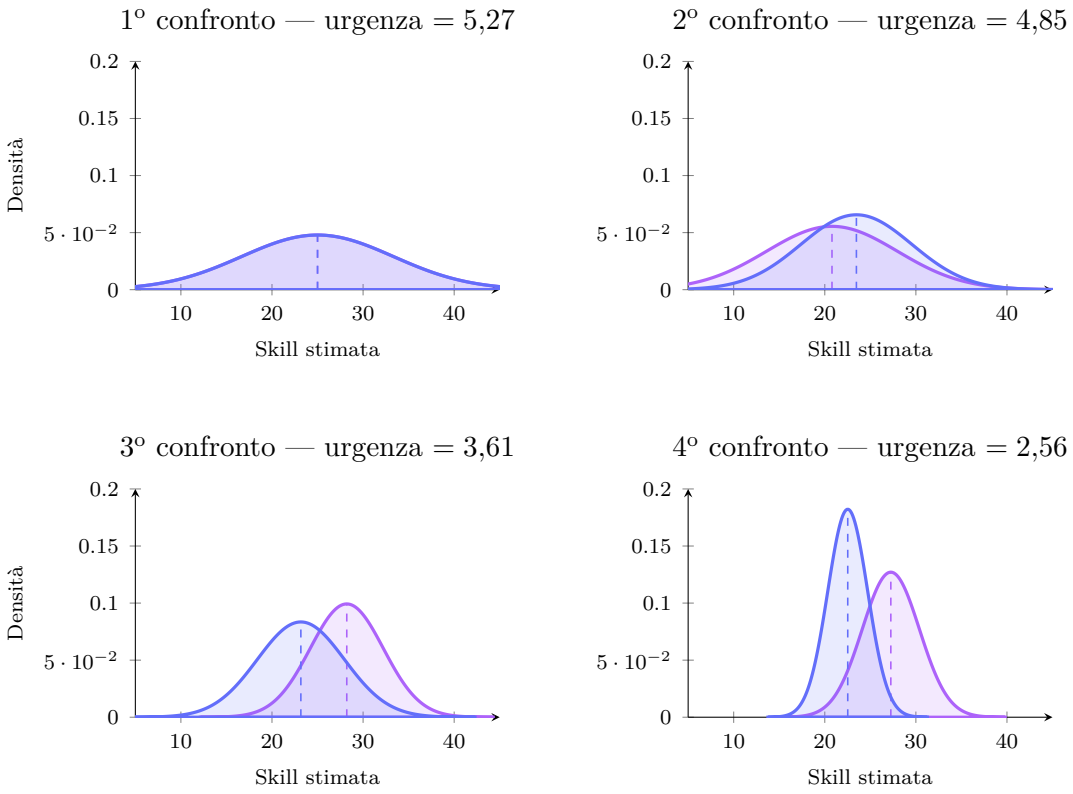


Figura B.2: Evoluzione di una stessa coppia (VoxCPM–Chatterbox, emozione *sorpresa*) lungo i suoi confronti successivi: le distribuzioni si restringono, le medie si separano e l'urgenza della coppia diminuisce a ogni voto.

Appendice C

Valori completi dei rating TrueSkill

La Tabella C.1 riporta, per ciascuno dei 28 player (coppia modello–emozione), la stima finale di abilità μ, la relativa incertezza σ e il numero di confronti n su cui la stima si basa. Questi valori sono quelli visualizzati in forma di distribuzione nella Figura 5.5.

Tabella C.1: Valori finali di abilità stimati da TrueSkill per tutti i 28 player

Emozione	**Modello**	μ	σ	n
Rabbia	Chatterbox	21,54	1,03	74
	OmniVoice	27,26	1,19	62
	VoxCPM	32,29	1,46	38
	Qwen3-TTS	18,37	1,10	66
Disgusto	Chatterbox	28,98	0,94	89
	OmniVoice	29,00	0,95	86
	VoxCPM	23,35	1,12	60
	Qwen3-TTS	19,86	1,11	63
Gioia	Chatterbox	22,19	0,99	81
	OmniVoice	26,96	1,36	45
	VoxCPM	33,00	1,73	30
	Qwen3-TTS	19,78	1,04	72
Paura	Chatterbox	23,75	1,18	56
	OmniVoice	32,77	1,19	57
	VoxCPM	28,70	1,18	58
	Qwen3-TTS	19,69	1,19	61
Tristezza	Chatterbox	20,25	0,96	82
	OmniVoice	31,10	1,72	29
	VoxCPM	24,38	1,35	45
	Qwen3-TTS	18,70	0,97	80
Neutro	Chatterbox	20,77	1,18	53
	OmniVoice	23,87	0,94	91
	VoxCPM	32,45	2,16	20
	Qwen3-TTS	25,04	0,97	80
Sorpresa	Chatterbox	22,98	1,23	53
	OmniVoice	27,84	1,30	47
	VoxCPM	33,23	1,35	46
	Qwen3-TTS	18,50	1,25	52

Ringraziamenti

A conclusione di questo lungo e intenso percorso di studi, desidero esprimere la mia gratitudine a coloro che mi hanno accompagnato in questo traguardo.

Il mio primo ringraziamento va alla mia famiglia, per avermi sostenuto in qualsiasi scelto e avermi dato l'opportunità di studiare lontano da casa.

Il ringraziamento più dolce va ad Aman. Grazie per aver creduto in me prima ancora che lo facessi io e per essere stato al mio fianco in ogni momento. È anche grazie a te se ho avuto l'audacia di trasferirmi in una nuova città e arrivare fin qui.

Un pensiero speciale è rivolto ai miei amici e compagni di corso: grazie per aver condiviso le fatiche dello studio, rendendo questi anni più leggeri e preziosi tra una risata e le immancabili pause caffè.

Un ringraziamento sincero va anche a Ciro, non solo per aver condiviso il lavoro di questi ultimi mesi, ma per la grande disponibilità e le tante risate che ci hanno accompagnato.

Infine, desidero ringraziare il Prof. Bottino, il Prof. Strada e Stefano per avermi seguita in questo percorso. Grazie per il tempo che mi avete dedicato e per avermi permesso di esplorare tematiche del tutto nuove per me, rendendo questo progetto una bellissima occasione di crescita.

Bibliografia

[1] Justine Cassell. «Embodied conversational interface agents». In: *Commun. ACM* 43.4 (apr. 2000), pp. 70–78. ISSN: 0001-0782. DOI: 10.1145/332051.332075. URL: https://doi.org/10.1145/332051.332075 (cit. a p. 1).

[2] S. Guo. «Developing a Scale for Measuring the Believability of Virtual Agents». In: *International Conference on Artificial Reality and Telexistence & Eurographics Symposium on Virtual Environments (ICAT-EGVE)*. Dublin, Ireland: Eurographics Digital Library, 2023, pp. 45–52. URL: https://par.nsf.gov/biblio/10499912 (cit. a p. 1).

[3] Joseph Bates. «The role of emotion in believable agents». In: *Commun. ACM* 37.7 (lug. 1994), pp. 122–125. ISSN: 0001-0782. DOI: 10.1145/176789.176803. URL: https://doi.org/10.1145/176789.176803 (cit. a p. 1).

[4] Andreas Triantafyllopoulos et al. «An Overview of Affective Speech Synthesis and Conversion in the Deep Learning Era». In: *Proceedings of the IEEE* 111.10 (ott. 2023), pp. 1355–1381. ISSN: 1558-2256. DOI: 10.1109/jproc.2023.3250266. URL: http://dx.doi.org/10.1109/JPROC.2023.3250266 (cit. a p. 1).

[5] Fabio Catania, Jordan Wilke e Franca Garzotto. «Emozionalmente: A Crowdsourced Corpus of Simulated Emotional Speech in Italian». In: *IEEE Transactions on Audio, Speech and Language Processing* PP (gen. 2025), pp. 1–14. DOI: 10.1109/TASLPRO.2025.3540662 (cit. a p. 1).

[6] Ralf Herbrich, Tom Minka e Thore Graepel. «TrueSkill™: A Bayesian Skill Rating System». In: *Advances in Neural Information Processing Systems 19 (NIPS 2006)*. MIT Press, 2006, pp. 569–576. URL: https://papers.neurips.cc/paper/3079-trueskilltm-a-bayesian-skill-rating-system.pdf (cit. alle pp. 2, 62, 68).

[7] Xu Tan, Tao Qin, Frank Soong e Tie-Yan Liu. *A Survey on Neural Speech Synthesis*. 2021. arXiv: 2106.15561 [eess.AS]. URL: https://arxiv.org/abs/2106.15561 (cit. a p. 3).

[8] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior e Koray Kavukcuoglu. *WaveNet: A Generative Model for Raw Audio*. 2016. arXiv: 1609.03499 [cs.SD]. URL: https://arxiv.org/abs/1609.03499 (cit. a p. 4).

[9] Yuxuan Wang et al. *Tacotron: Towards End-to-End Speech Synthesis*. 2017. arXiv: 1703.10135 [cs.CL]. URL: https://arxiv.org/abs/1703.10135 (cit. a p. 4).

[10] Daniel Griffin e Jae Lim. «Signal Estimation from Modified Short-Time Fourier Transform». In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 32.2 (1984), pp. 236–243. DOI: 10.1109/TASSP.1984.1164317 (cit. a p. 5).

[11] Jonathan Shen et al. *Natural TTS Synthesis by Conditioning WaveNet on Mel Spectrogram Predictions*. 2018. arXiv: 1712.05884 [cs.CL]. URL: https://arxiv.org/abs/1712.05884 (cit. a p. 5).

[12] Yi Ren, Yangjun Ruan, Xu Tan, Tao Qin, Sheng Zhao, Zhou Zhao e Tie-Yan Liu. *FastSpeech: Fast, Robust and Controllable Text to Speech*. 2019. arXiv: 1905.09263 [cs.CL]. URL: https://arxiv.org/abs/1905.09263 (cit. a p. 5).

[13] Jungil Kong, Jaehyeon Kim e Jaekyoung Bae. *HiFi-GAN: Generative Adversarial Networks for Efficient and High Fidelity Speech Synthesis*. 2020. arXiv: 2010.05646 [cs.SD]. URL: https://arxiv.org/abs/2010.05646 (cit. a p. 5).

[14] Jaehyeon Kim, Jungil Kong e Juhee Son. *Conditional Variational Autoencoder with Adversarial Learning for End-to-End Text-to-Speech*. 2021. arXiv: 2106.06103 [cs.SD]. URL: https://arxiv.org/abs/2106.06103 (cit. a p. 6).

[15] Zalán Borsos et al. *AudioLM: a Language Modeling Approach to Audio Generation*. 2023. arXiv: 2209.03143 [cs.SD]. URL: https://arxiv.org/abs/2209.03143 (cit. a p. 6).

[16] Chengyi Wang et al. *Neural Codec Language Models are Zero-Shot Text to Speech Synthesizers*. 2023. arXiv: 2301.02111 [cs.CL]. URL: https://arxiv.org/abs/2301.02111 (cit. a p. 6).

[17] Eugene Kharitonov, Damien Vincent, Zalán Borsos, Raphaël Marinier, Sertan Girgin, Olivier Pietquin, Matt Sharifi, Marco Tagliasacchi e Neil Zeghidour. *Speak, Read and Prompt: High-Fidelity Text-to-Speech with Minimal Supervision*. 2023. arXiv: 2302.03540 [cs.SD]. URL: https://arxiv.org/abs/2302.03540 (cit. a p. 7).

[18] Mateusz Łajszczak et al. *BASE TTS: Lessons from building a billion-parameter Text-to-Speech model on 100K hours of data*. 2024. arXiv: 2402.08093 [cs.LG]. URL: https://arxiv.org/abs/2402.08093 (cit. a p. 8).

[19] Zhihao Du et al. *CosyVoice: A Scalable Multilingual Zero-shot Text-to-speech Synthesizer based on Supervised Semantic Tokens*. 2024. arXiv: 2407.05407 [cs.SD]. URL: https://arxiv.org/abs/2407.05407 (cit. a p. 8).

[20] Robert M. Gray. «Vector Quantization». In: *IEEE ASSP Magazine* 1.2 (apr. 1984), pp. 4–29. DOI: 10.1109/MASSP.1984.1162229 (cit. a p. 8).

[21] Biing-Hwang Juang e Augustine H. Gray. «Multiple Stage Vector Quantization for Speech Coding». In: *ICASSP '82. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*. Vol. 7. 1982, pp. 597–600. DOI: 10.1109/ICASSP.1982.1171604 (cit. a p. 9).

[22] Aaron van den Oord, Oriol Vinyals e Koray Kavukcuoglu. *Neural Discrete Representation Learning*. 2018. arXiv: 1711.00937 [cs.LG]. URL: https://arxiv.org/abs/1711.00937 (cit. a p. 10).

[23] Neil Zeghidour, Alejandro Luebs, Ahmed Omran, Jan Skoglund e Marco Tagliasacchi. *SoundStream: An End-to-End Neural Audio Codec*. 2021. arXiv: 2107.03312 [cs.SD]. URL: https://arxiv.org/abs/2107.03312 (cit. a p. 10).

[24] Alexandre Défossez, Jade Copet, Gabriel Synnaeve e Yossi Adi. *High Fidelity Neural Audio Compression*. 2022. arXiv: 2210.13438 [eess.AS]. URL: https://arxiv.org/abs/2210.13438 (cit. a p. 10).

[25] ONNX. *ONNX: Open Neural Network Exchange*. https://onnx.ai (cit. a p. 13).

[26] Unity Technologies. *Unity Sentis*. https://docs.unity3d.com/Packages/com.unity.ai.inference@latest (cit. a p. 13).

[27] ONNX Runtime developers. *ONNX Runtime*. https://onnxruntime.ai/. Version: x.y.z. 2021 (cit. alle pp. 15, 31).

[28] lookbe. *Orpheus-TTS-Unity: High-Performance Local Neural Text-to-Speech for Unity*. https://github.com/lookbe/orpheus-tts-unity. GitHub repository. 2025 (cit. a p. 15).

[29] ElevenLabs. *ElevenLabs: AI Voice Generator and Text to Speech*. https://elevenlabs.io (cit. alle pp. 15, 64).

[30] Microsoft. *Azure AI Speech: Text to Speech*. https://azure.microsoft.com/products/ai-services/ai-speech (cit. a p. 15).

[31] Google. *Google Cloud Text-to-Speech*. https://cloud.google.com/text-to-speech (cit. a p. 15).

[32] Amazon Web Services. *Amazon Polly*. https://aws.amazon.com/polly/ (cit. a p. 15).

[33] OpenAI. *OpenAI: Text to Speech*. https://platform.openai.com/docs/guides/text-to-speech (cit. a p. 15).

[34] Hangrui Hu et al. *Qwen3-TTS Technical Report*. 2026. arXiv: 2601.15621 [cs.SD]. URL: https://arxiv.org/abs/2601.15621 (cit. alle pp. 16, 19–21, 25).

[35] Han Zhu et al. *OmniVoice: Towards Omnilingual Zero-Shot Text-to-Speech with Diffusion Language Models*. 2026. arXiv: 2604.00688 [cs.CL]. URL: https://arxiv.org/abs/2604.00688 (cit. a p. 16).

[36] Yixuan Zhou et al. «VoxCPM: Tokenizer-Free TTS for Context-Aware Speech Generation and True-to-Life Voice Cloning». In: *arXiv preprint arXiv:2509.24650* (2025) (cit. a p. 16).

[37] Resemble AI. *Chatterbox-TTS*. https://github.com/resemble-ai/chatterbox. GitHub repository. 2025 (cit. a p. 16).

[38] MiniMax. *MiniMax Audio: Text to Speech*. https://www.minimax.io/audio (cit. a p. 19).

[39] Alexandre Défossez, Laurent Mazaré, Manu Orsini, Amélie Royer, Patrick Pérez, Hervé Jégou, Edouard Grave e Neil Zeghidour. «Moshi: a speech-text foundation model for real-time dialogue». In: *arXiv preprint arXiv:2410.00037* (2024) (cit. a p. 21).

[40] Sanyuan Chen, Chengyi Wang et al. «WavLM: Large-Scale Self-Supervised Pre-Training for Full Stack Speech Processing». In: *IEEE Journal of Selected Topics in Signal Processing* (2022) (cit. a p. 21).

[41] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen e Yunfeng Liu. «RoFormer: Enhanced Transformer with Rotary Position Embedding». In: *arXiv preprint arXiv:2104.09864* (2021) (cit. a p. 22).

[42] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón e Sumit Sanghai. «GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints». In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2023, pp. 4895–4901 (cit. a p. 23).

[43] Rithesh Kumar, Prem Seetharaman, Alejandro Luebs, Ishaan Kumar e Kundan Kumar. «High-Fidelity Audio Compression with Improved RVQGAN». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 36. 2023, pp. 27980–27993 (cit. a p. 27).

[44] Brecht Desplanques, Jenthe Thienpondt e Kris Demuynck. «ECAPA-TDNN: Emphasized Channel Attention, Propagation and Aggregation in TDNN Based Speaker Verification». In: *Proc. Interspeech*. 2020, pp. 3830–3834 (cit. a p. 27).

[45] PyTorch. *Exporting a Model from PyTorch to ONNX and Running it using ONNX Runtime*. `https://docs.pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html`. Tutorial ufficiale PyTorch. Consultato il 13 Luglio 2026 (cit. alle pp. 30, 31).

[46] ElBruno. *ElBruno.QwenTTS*. `https://github.com/elbruno/ElBruno.QwenTTS`. Consultato il 10 Luglio 2026. 2026 (cit. alle pp. 31, 33, 41).

[47] zukky. *Qwen3-TTS-ONNX-DLL*. `https://huggingface.co/zukky/Qwen3-TTS-ONNX-DLL`. Repository Hugging Face, derivato da Qwen3-TTS. Consultato il: 2026-07-04. 2026 (cit. a p. 52).

[48] rekuenkdr. *Qwen3-TTS-streaming: real-time streaming inference for Qwen3-TTS*. `https://github.com/rekuenkdr/Qwen3-TTS-streaming`. repository software, consultato il 9 luglio 2026. 2026 (cit. a p. 61).

[49] Paul Ekman. «An Argument for Basic Emotions». In: *Cognition and Emotion* 6.3–4 (1992), pp. 169–200. DOI: 10.1080/02699939208411068 (cit. a p. 63).

[50] Keisuke Sakaguchi, Matt Post e Benjamin Van Durme. «Efficient Elicitation of Annotations for Human Evaluation of Machine Translation». In: *Proceedings of the Ninth Workshop on Statistical Machine Translation*. A cura di Ondřej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Christof Monz, Matt Post e Lucia Specia. Baltimore, Maryland, USA: Association for Computational Linguistics, giu. 2014, pp. 1–11. DOI: 10.3115/v1/W14-3301. URL: https://aclanthology.org/W14-3301/ (cit. alle pp. 66, 77).

[51] ITU-T. *Recommendation P.800: Methods for Subjective Determination of Transmission Quality*. Recommendation P.800. International Telecommunication Union, 1996. URL: https://www.itu.int/rec/T-REC-P.800-199608-I/en (cit. a p. 66).

[52] Arpad E. Elo. *The Rating of Chessplayers, Past and Present*. New York: Arco Pub., 1978. ISBN: 9780668047210. URL: http://www.amazon.com/Rating-Chess-Players-Past-Present/dp/0668047216 (cit. a p. 67).

[53] Aliaksei Mikhailiuk, Clifford Wilmot, Maria Perez-Ortiz, Dingcheng Yue e Rafał K. Mantiuk. «Active Sampling for Pairwise Comparisons via Approximate Message Passing and Information Gain Maximization». In: *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 2559–2566. URL: https://arxiv.org/abs/2004.05691 (cit. alle pp. 68, 77).

[54] Jing Li, Rafał Mantiuk, Junle Wang, Suiyi Ling e Patrick Le Callet. «Hybrid-MST: A Hybrid Active Sampling Strategy for Pairwise Preference Aggregation». In: *Advances in Neural Information Processing Systems 31 (NeurIPS)*. 2018. URL: `https://proceedings.neurips.cc/paper/2018/hash/8b6a80c3cf2cbd5f967063618dc54f39-Abstract.html` (cit. alle pp. 68, 77).