



Università Politecnica Delle Marche

Dipartimento di Ingegneria dell'Informazione

LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA E
DELL'AUTOMAZIONE

Parola di Motoneurone:

**Progettazione e sviluppo di una soluzione basata su cloud computing
per monitorare l'andamento della disartria nei pazienti affetti da
malattie neurologiche**

**Design and development of a solution based on cloud computing to
monitor the evolution of dysarthria in patients affected by
neurological disorders**

Relatore:

Dott. Adriano Mancini

Correlatori:

Prof. Emanuele Frontoni

Dott.ssa Lucia Migliorelli

Candidato: 1096272

Kevin Cela

ANNO ACCADEMICO 2020 / 2021

Sommario

La presente tesi descrive il processo di progettazione e sviluppo della componente architettonica e di *backend* del progetto Parola di Motoneurone.

Tale progetto, promosso dall'Università Politecnica delle Marche e vincitore del bando Fondazione SIN 2020 proposto dalla Società Italiana di Neurologia (SIN) e dall'azienda americana Biogen, è un sistema che ha l'obiettivo di analizzare dati multimediali (audio e video) registrati da persone affette da Sclerosi Laterale Amiotrofica (SLA), al fine di monitorare l'andamento della disartria e agire tempestivamente nel caso di una rapida evoluzione.

Tramite questo sistema sarà quindi possibile effettuare il monitoraggio di parametri critici per il paziente da remoto individuando eventualmente dei *trend*, senza la necessità, da parte del paziente, di eseguire esercizi direttamente in presenza e senza utilizzare sensoristica "ingombrante": lo svolgimento degli esercizi, infatti, avviene esclusivamente tramite registrazioni video oppure audio, provenienti da un dispositivo mobile quale smartphone o tablet.

Un altro vantaggio di questo sistema, inoltre, consiste nell'utilizzo di scale di valutazione quantitative piuttosto che qualitative: mentre le scale tradizionalmente usate per la valutazione della disartria si basano su parametri qualitativi, e quindi soggettivi, il sistema utilizzerà degli algoritmi di analisi che individueranno degli indici oggettivi, non dipendenti dalla persona singola e quindi più facili da impiegare e confrontare.

La tesi, in questo contesto, si occuperà di esporre nel dettaglio il background clinico che ha portato alla proposta di questo progetto, così come i processi relativi alla progettazione e all'implementazione della componente architettonica

e di *backend* del sistema, che ha l'obiettivo di essere facilmente estensibile anche per l'analisi di altre patologie, oltre alla SLA (patologia presa in considerazione per l'implementazione iniziale).

Nel primo capitolo verrà descritto il background clinico, necessario al lettore per capire le necessità che hanno portato alla proposta del progetto Parola di Motoneurone, che verrà quindi presentato nel dettaglio, assieme agli obiettivi che intende soddisfare.

Nel secondo capitolo verranno presentate le tecnologie e gli strumenti che verranno usati nell'ambito del progetto, al fine di far acquisire familiarità al lettore relativamente a questi, per una migliore comprensione dei capitoli successivi.

Nel terzo capitolo si passerà quindi alla fase di progettazione del sistema in cui, definiti i requisiti funzionali e non funzionali, verranno illustrati i passi relativi alla progettazione del *backend* e della componente architetturale/cloud del sistema, con un focus sull'ingestione e sull'elaborazione offline automatica dei dati multimediali dei pazienti. In questo capitolo verrà anche motivata la scelta delle tecnologie descritte nel secondo capitolo.

Nel quarto capitolo, quindi, si descriverà nel dettaglio l'implementazione della soluzione progettata nel capitolo precedente, con un focus sulle soluzioni adottate al fine di avere un sistema facilmente estendibile anche per l'uso di algoritmi e metriche di analisi aggiuntivi.

Negli ultimi due capitoli, infine, verranno presentati alcuni dei risultati così ottenuti e saranno tratte delle conclusioni, descrivendo eventuali sviluppi futuri.

Indice

1	Introduzione e background clinico	9
1.1	La disartria e il suo ruolo nelle malattie neurologiche	9
1.2	Monitoraggio della disartria nei pazienti con patologie neurodegenerative	10
1.3	Presentazione progetto Parola di Motoneurone	14
2	Strumenti e metodi utilizzati	18
2.1	Amazon Web Services	18
2.1.1	Amazon S3	18
2.1.2	AWS Lambda	19
2.1.3	API Gateway	20
2.1.4	Amazon Simple Queue System	21
2.1.5	Amazon Cognito	21
2.1.6	Amazon Virtual Private Cloud	22
2.1.7	Amazon Relational Database Service	23
2.1.8	AWS CloudFormation e AWS Cloud Development Kit	23
2.2	PostgreSQL	24
2.3	GraphQL	25
2.4	Ambienti e linguaggi di programmazione usati	26
3	Parola di motoneurone: progettazione dell'architettura cloud e del database	27
3.1	Descrizione progetto	27
3.1.1	Specifiche dell'applicazione del paziente	28

3.1.2	Specifiche dell'applicazione del clinico	30
3.2	Analisi dei requisiti	33
3.2.1	Requisiti funzionali	33
3.2.2	Requisiti non funzionali	35
3.3	Progettazione dell'architettura cloud	36
3.3.1	Scelta del servizio di storage per il salvataggio degli esercizi	36
3.3.2	Interfacciamento dell'applicazione del paziente con lo storage	36
3.3.3	Scelta della componente per l'esecuzione degli algoritmi di analisi	38
3.3.4	Distribuzione automatica degli esercizi per ciascun algoritmo	40
3.3.5	Gestione dell'autenticazione del paziente e del clinico . . .	43
3.3.6	Architettura cloud	44
3.4	Scelta del database e progettazione del modello di dati	45
3.4.1	Progettazione concettuale	45
3.4.2	Scelta del modello logico e del database	48
3.4.3	Progettazione logica e fisica	52
4	Sviluppo del progetto	54
4.1	Sviluppo infrastruttura	54
4.2	Sviluppo delle funzioni lambda	60
4.2.1	Lambda per il caricamento degli esercizi su bucket	60
4.2.2	Lambda orchestratore	62
4.2.3	Lambda consumatore	65
4.3	Sviluppo del backend	67
4.3.1	Migrazione e seeding del database	69
4.3.2	Sviluppo dei modelli del backend	72
4.3.3	Generazione automatica degli schemi dai modelli	74
5	Risultati	83
6	Conclusione e sviluppi futuri	87
6.1	Sviluppi futuri	88

Bibliografia

95

Capitolo 1

Introduzione e background clinico

In questo capitolo verranno introdotti alcuni concetti relativi alle metodologie tradizionalmente utilizzate per valutare l'evoluzione di una malattia neurologica, tramite il monitoraggio della disartria del paziente. Si illustreranno quindi le principali problematiche relative agli approcci utilizzati in letteratura che porteranno all'introduzione del progetto Parola di Motoneurone, di cui verranno illustrate le specifiche e gli obiettivi che intende soddisfare.

1.1 La disartria e il suo ruolo nelle malattie neurologiche

La disartria è un disturbo motorio del linguaggio che deriva da una lesione di tipo neurologico che coinvolge la componente motoria del linguaggio, ed è generalmente caratterizzata da una scarsa capacità di articolazione dei fonemi, con un conseguente deficit nel linguaggio e nella intelligibilità del parlato.

La disartria può essere differenziata in sei tipologie principali: disartria flaccida, spastica, atassica, ipocinetica, ipercinetica e mista, ognuna delle quali legata ad una specifica disfunzione motoria/cerebrale/cerebellare [1].

In molte patologie neurologiche, quali ad esempio la sclerosi laterale amiotrofica (SLA) bulbare o il morbo di Parkinson, la disartria è uno dei sintomi e segni di esordio più importanti, e una sua attenta analisi può essere di fondamentale importanza per definire lo stadio di avanzamento della malattia. Ad esempio, considerando la SLA bulbare, lo studio della disartria del paziente nel tempo consente di individuare il momento opportuno per indicare strategie di compenso alla disabilità comunicativa, di monitorare l'andamento della malattia e di ottenere una misura di *outcome* nei *trial* clinici [2].

1.2 Monitoraggio della disartria nei pazienti con patologie neurodegenerative

La valutazione della disartria nei pazienti affetti da malattie neurologiche avviene generalmente mediante l'uso di specifiche scale di valutazione ed autovalutazione. Tra tutte, una delle scale più usate è il profilo Robertson [3].

Il profilo Robertson [3] è una scala non sommativa per la valutazione clinica della disartria, composta da 71 *item* che valutano vari aspetti clinici correlati con l'articolazione verbale. Essendo una scala non sommativa, i 71 *item* si comportano come se fossero scale *mono-item*.

Gli *item* sono raggruppati nelle seguenti sottocategorie:

- fonazione;
- respirazione;
- muscolatura bucco-facciale;
- diadococinesi;
- riflessi;
- articolazioni;
- intelligibilità;
- prosodia.

Nella Figura 1.1, in particolare, è possibile visionare alcuni degli item relativi alla scala, usati per la valutazione della disartria.

Oltre alla scala di Robertson, viene somministrata al paziente la compilazione di una scheda di autovalutazione, che consente l'ottenimento di una valutazione soggettiva delle caratteristiche della parola e delle difficoltà di eloquio nelle situazioni sociali, nonché le strategie di compenso adottate e le relazioni degli interlocutori. Nella Figura 1.2 è possibile vedere alcuni degli item di questa scala di autovalutazione.

ITEM		1	2	3	4	
Muscolatura facciale	1. Simmetria facciale a riposo	a) Faccia				
	2. Cambio di espressione su sorriso					
	3. Mantenimento chiusura labiale a riposo	b) Labbra				
	4. Stiramento labbra					
	5. Protrusione labbra					
	6. Chiusura labiale durante la conversazione					
	7. Bocca aperta / bocca chiusa	c) Mandibola				
	8. Lateralizzazione mandibola a destra					
	9. Lateralizzazione mandibola a sinistra					
	10. Protrudere la lingua	d) Lingua				
	11. Retrarre la lingua					
	12. Muovere la lingua a destra					
	13. Muovere la lingua a sinistra					
	14. Passare la lingua sopra i denti					
	15. Punta lingua contro guancia destra					
	16. Punta lingua contro guancia sinistra					
	17. Alzare punta lingua nella bocca					
	18. Alzare punta lingua fuori della bocca					
	19. Elevazione velo del palato su /a/		e) Velo			
	20. Elevazione velo del palato su serie di /a/					
ITEM		1	2	3	4	
Diadococinesi	1. Aprire e chiudere la bocca rapidamente N° in 5 sec.:					
	2. Protrudere e retrarre le labbra rapidamente N° in 5 sec.:					
	3. Protrudere e retrarre la lingua rapidamente N° in 5 sec.:					
	4. Alzare e abbassare punta lingua rapidamente N° in 5 sec.:					
	5. Muovere la lingua da parte a parte rapidamente N° in 5 sec.:					
	6. Ripetere "u-i" rapidamente N° in 5 sec.:					
	7. Ripetere "pa..." rapidamente N° in 5 sec.:					
	8. Ripetere "ta..." rapidamente N° in 5 sec.:					
	9. Ripetere "ka..." rapidamente N° in 5 sec.:					
	10. Ripetere "kala ..." rapidamente N° in 5 sec.:					
	11. Ripetere "p.t.k...."rapidamente N° in 5 sec.:					
ITEM		1	2	3	4	
Fonazione	1 Attacco su /a/					
	2. Emissione /a/ prolungata					
	3. Emissione/a/ intensità elevata					
	4. Crescendo su /a/					
	5. Diminuendo su /a/					
	6. Serie di /a/					
	7. Scala ascendente su /a/					
	8. Scala discendente su /a/					
	9. Glissato ascendente su /a/					
	10. Glissato discendente su /a/					
	11. Adeguata intensità di conversazione					
	12. Qualità vocale					

Figura 1.1: Alcuni item della scala di valutazione del profilo Robertson. Si noti come la scala è di natura qualitativa, presentando soltanto quattro possibili valutazioni: Scarso (1), Discreto (2), Buono (3), Ottimo (4). [3]

COGNOME _____ NOME _____ DATA _____

Questionario di Autovalutazione della Disartria
(Yorkston, Bombardier e Hammen, 1994; vers.it. Schindler e Gulli, 2002)

Item	Mai	Quasi mai	Qualche volta	Quasi sempre	Sempre
Caratteristiche della parola					
1. La mia voce suona rauca e aspra					
2. Il mio modo di parlare è lento					
3. Il mio modo di parlare è troppo forte o troppo debole					
4. Il mio modo di parlare ha una qualità nasale					
5. Ho difficoltà a parlare quando sono di fretta					
6. Quando sono stanco/a parlo peggio					
7. Quando parlo con estranei hanno difficoltà a comprendermi					
8. I miei familiari hanno difficoltà a comprendermi					
Strategie di compenso					
1. Faccio "tradurre" quello che dico ad amici o familiari quando gli estranei non mi capiscono					
2. Quando parlo con le persone mi assicuro che mi vedano bene (ad es. sto in luoghi ben illuminati o mi assicuro che siano di fronte a me)					
3. Se qualcuno non ha capito o ha capito male quello che ho detto, ripeto il messaggio usando altre parole					
4. Sottolineo l'argomento della conversazione in modo tale che il mio interlocutore mi capisca meglio					
5. Quando qualcuno non ha capito ripeto più forte o più lentamente					
6. Dico agli altri di segnalarmi quando hanno difficoltà a capirmi					
7. Evito di parlare con estranei a causa dei miei problemi di espressione					
8. Quando sono coinvolto/a in una importante conversazione, spengo la radio, la TV o altre sorgenti di rumore					
Totali:					

Figura 1.2: Alcuni item della scala di autovalutazione della disartria [3]

In letteratura la maggior parte delle valutazioni della disartria nel paziente si basa su valutazioni qualitative come le scale sopra descritte, che spesso sono limitate all'analisi delle sole caratteristiche vocali [4]. Questi approcci, seppur possono effettivamente portare a una quantificazione dell'evoluzione della patologia nel paziente, presentano diversi problemi in un contesto applicativo, sia dal punto di vista del paziente che del clinico. Si ha infatti, dal punto di vista del clinico, che le scale utilizzate non sono quantificabili in maniera esatta in quanto sono di natura qualitativa, con la conseguenza che la valutazione del progresso della patologia potrebbe essere influenzata da aspetti soggettivi legati alla metodologia di valutazione del singolo clinico. Oltre a ciò, è necessario inoltre

considerare la necessità, da parte del paziente, di doversi presentare fisicamente dal clinico per svolgere gli esercizi richiesti per l'ottenimento delle valutazioni. Questo fatto è particolarmente problematico nel caso di pazienti con disabilità motoria che trovano difficoltà nello spostarsi verso centri dislocati nel territorio, o di pazienti con gravi disabilità che necessitano di valutazioni più frequenti, dovendo dunque recarsi frequentemente nel centro di riferimento. È infine importante aggiungere il fatto che il profilo di Robertson, generalmente, ha dei tempi di somministrazione non inferiori ad 1 ora, nonostante i diversi tentativi di ridurre la complessità e gli *item* investigativi.

Un'alternativa all'uso delle scale qualitative consiste nell'adozione di sensori, che consentono di effettuare accurate analisi in tempo reale. Alcuni dei sensori che vengono utilizzati sono per esempio reti palatali, visibili in Figura 1.3, che però raramente si prestano ad una continuativa azione di monitoraggio durante la pratica clinica [5]. La natura fisica ed ingombrante della sensoristica infatti, seppur miniaturizzata, provoca disagio nel paziente interferendo significativamente con le sue azioni. Inoltre, tale sensoristica necessita, per una corretta fruizione, di operazioni di costante manutenzione, personalizzazione e ricalibrazione dei dispositivi, che non consentono una adeguata scalabilità nell'uso continuativo di questi strumenti per un monitoraggio efficace e continuativo del paziente.

1.3 Presentazione progetto Parola di Motoneurone

I problemi nell'uso delle scale qualitative e dei sensori menzionati nella sezione precedente hanno portato alla formulazione del progetto Parola di Motoneurone, che fornisce una soluzione a queste problematiche fornendo una piattaforma di monitoraggio da remoto con la possibilità di ottenere indici oggettivi.

Il progetto Parola di Motoneurone mira a realizzare un sistema che monitori in maniera oggettiva, non invasiva e da remoto la progressione della disartria nei pazienti affetti da malattie neurologiche. In particolare, si considereranno inizialmente pazienti affetti da SLA bulbare, ma successivamente si intende espandere il campo applicativo del sistema anche per altre patologie che necessitano

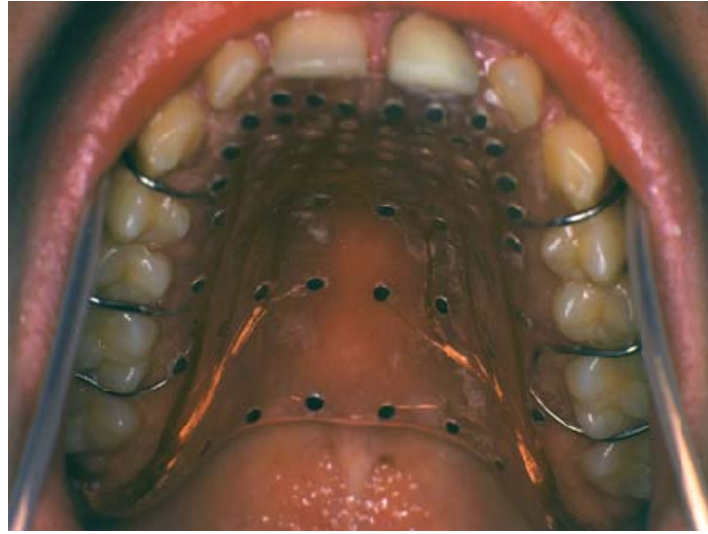


Figura 1.3: Sensori utilizzati per l'elettropalatografia, per registrare il comportamento della lingua durante l'eloquio del paziente. L'elettropalatografia combinata alla logopedia convenzionale sono utili nel trattamento delle disartrie moderate e gravi, ma la sensoristica deve essere realizzata ad-hoc per ogni paziente, riducendo la scalabilità e le tempistiche di intervento [6].

un monitoraggio dell'evoluzione della malattia e dei progressi, o delle difficoltà, riscontrate dal paziente, tramite l'analisi della disartria.

Parola di Motoneurone garantirà ai clinici uno strumento di supporto utile ad identificare tempestivamente i cambiamenti funzionali nelle performance dei pazienti e ad accelerare l'attuazione di eventuali strategie correttive e compensatorie. Al paziente basterà scaricare una applicazione mobile (alcuni dei suoi *mockup* sono visibili in Figura 1.4) che, mantenendo accesi videocamera e microfono del *device*, lo guiderà nell'esecuzione di compiti vocali (ad esempio vocalizzazioni, ripetizioni di sillabe in sequenza) o di azioni precise (ad esempio aprire e chiudere la bocca velocemente). Assieme al progetto verranno sviluppati algoritmi di intelligenza artificiale che analizzano i dati audio e video delle singole sessioni, e restituiranno al clinico, tramite una *dashboard* semplice da consultare (in Figura 1.5 è presente una schermata del prototipo dell'applicazione, attualmente in sviluppo), tutte le variabili rilevanti per valutare l'andamento della disartria.

Gli obiettivi clinici del progetto Parola di Motoneurone sono:

- Monitorare da remoto l'evoluzione della SLA bulbare tramite la valutazione

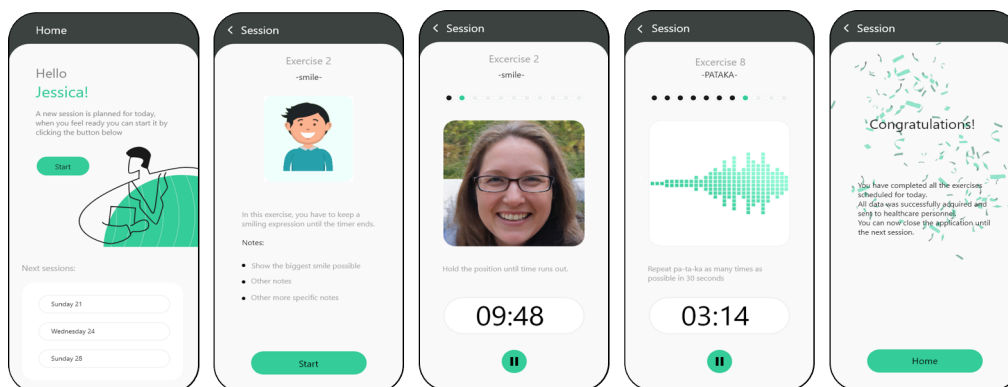


Figura 1.4: Mockup dell'applicazione lato paziente del progetto Parola di Motoneurone

rapida e differita del paziente sulla base di misure quantitative relative all'andamento della disartria.

- Personalizzare l'approccio assistenziale al paziente affetto da SLA bulbare sulla base delle misure quantitative fornite dal sistema di monitoraggio.

Parola di Motoneurone si pone inoltre come obiettivo clinico secondario quello di valutare la correlazione fra i dati oggettivi relativi all'evoluzione della disartria e dati inerenti altri ambiti di interessamento bulbare (disfagia). Tale caratteristica permetterà al clinico di monitorare da remoto tutti i segnali di allarme del peggioramento dei sintomi per cui è necessario attivare una valutazione immediata.

Gli obiettivi scientifici e tecnologici del progetto Parola di Motoneurone, invece, sono di progettare e sviluppare:

- una soluzione che sia utile alla raccolta e all'archiviazione dei dati risultanti dalle analisi degli algoritmi e che contemporaneamente ne garantisca sempre l'integrità (privacy, sicurezza e protezione del dato).
- una soluzione che sia accettata dai pazienti oggetto dello studio e che sia facilmente utilizzabile sia dai pazienti che dai clinici.

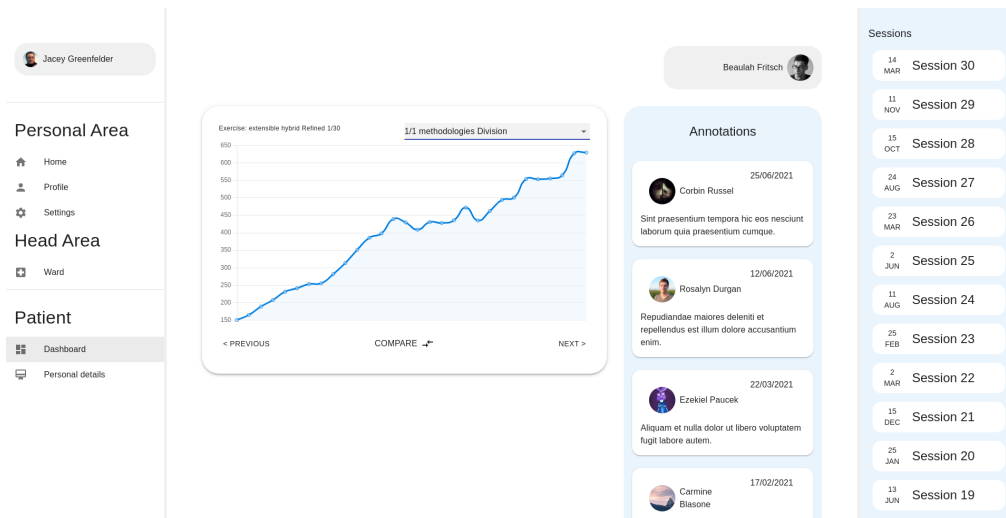


Figura 1.5: Prototipo della dashboard dell'applicazione web lato clinico del progetto Parola di Motoneurone

- algoritmi di intelligenza artificiale che estrapolino automaticamente dalle sequenze audio/video grandezze quantitative utili al monitoraggio della disartria.

Nel contesto di Parola di Motoneurone, la presente tesi si occuperà di illustrare la progettazione e la realizzazione della componente di *backend* ed architetturale del sistema. Il sistema dovrà essere in grado di:

1. ingerire informazioni multimediali, provenienti dall'applicazioni mobile del paziente;
2. Supportare l'elaborazione delle informazioni multimediali in maniera automatica tramite appositi algoritmi di intelligenza artificiale;
3. tradurre le predizioni degli algoritmi in informazione clinica di interesse per valutare l'evoluzione della malattia;
4. memorizzare i dati in un database per permettere al clinico la loro visualizzazione tramite la *dashboard* dedicata.

Capitolo 2

Strumenti e metodi utilizzati

In questo capitolo saranno brevemente presentati gli strumenti tecnologici utilizzati per la realizzazione del progetto, al fine di introdurre il lettore alle tecnologie di cui dopo si farà uso nei successivi capitoli di progettazione ed implementazione.

2.1 Amazon Web Services

Amazon Web Services (AWS) [7] è la piattaforma di Amazon che fornisce servizi di *cloud computing* di varia natura. Questi servizi sono operativi in 20 regioni geografiche in cui Amazon stessa ha suddiviso il globo. AWS offre oltre 150 servizi, fornendo soluzioni on-demand con caratteristiche di alta disponibilità, ridondanza e sicurezza, in cui il costo finale deriva dalla combinazione di tipo e quantità di risorse utilizzate, caratteristiche scelte dall'utente, tempo di utilizzo e performance desiderate. AWS è quindi, di fatto, una delle piattaforme principali per quanto riguarda il *cloud computing*.

Le tecnologie e i servizi di AWS saranno utilizzati nel contesto del progetto di tesi per la progettazione dell'architettura e l'implementazione del sistema.

2.1.1 Amazon S3

AWS S3 è il principale servizio di *storage* web offerto da AWS che offre scalabilità, disponibilità dei dati, sicurezza e prestazioni comparabili allo stato dell'arte del settore. S3 offre caratteristiche di gestione semplici da utilizzare, che consentono

di organizzare i dati e di configurare controlli di accesso ottimizzati per soddisfare requisiti aziendali, di pianificazione e di conformità specifici, garantendo eventualmente anche la possibilità di usare algoritmi di crittografia *server-side* per cifrare i dati, con garanzie di integrità e riservatezza.

La memorizzazione dei dati in S3 avviene tramite l'utilizzo dei cosiddetti *bucket*, che rappresentano un generico contenitore all'interno del quale è possibile effettuare l'*upload* di file. Quando si archiviano dei dati, si assegna loro una chiave oggetto unica che serve per recuperarli in seguito. Le chiavi possono essere costituite da qualsiasi stringa e possono essere formulate in modo da riprodurre attributi gerarchici, quali ad esempio cartelle.

Amazon S3 offre automaticamente una consistenza forte per la lettura e la scrittura, senza modifiche alle prestazioni o alla disponibilità, senza sacrificare l'isolamento regionale per le applicazioni e senza costi aggiuntivi. Dopo la scrittura di un nuovo oggetto o la sovrascrittura di un oggetto esistente, ogni richiesta di lettura successiva riceve immediatamente l'ultima versione dell'oggetto. S3 fornisce anche una consistenza forte per le operazioni di elenchi, così dopo la scrittura è possibile eseguire immediatamente un elenco degli oggetti in un *bucket* con le eventuali modifiche applicate.

Nel contesto del progetto, Amazon S3 verrà principalmente usato come per memorizzare i dati multimediali relativi agli esercizi svolti dai pazienti, che verranno memorizzati sotto forma di file compressi e che saranno successivamente elaborati dagli algoritmi di analisi per l'ottenimento degli indici oggettivi.

2.1.2 AWS Lambda

Molte delle funzionalità chiave del progetto verranno implementate in un contesto *serverless*, facendo uso del servizio AWS Lambda.

AWS Lambda è uno dei servizi più utilizzati di AWS, che consente di eseguire codice senza dover effettuare il *provisioning* né gestire server. Il costo viene determinato sulla base dei tempi di elaborazione, perciò non viene addebitato alcun costo quando il codice non è in esecuzione, rendendo il servizio particolarmente adatto per applicazioni che vengono usate per brevi intervalli di tempo,

e non in maniera continuata. Una volta caricato il codice, Lambda si prende carico delle azioni necessarie per eseguirlo e ricalibrarne le risorse con la massima disponibilità. È inoltre possibile configurare il servizio in modo tale che il codice associato venga attivato automaticamente da altri servizi AWS oppure che venga richiamato direttamente da qualsiasi app Web o mobile. Questo è molto utile, per esempio, per avviare il servizio ogni qualvolta un file viene caricato su un *bucket* S3.

AWS supporta diversi *runtime* per l'esecuzione del codice: sono disponibili infatti degli ambienti NodeJS, così come degli ambienti Python, oppure anche Java, per l'esecuzione del codice [8]. Bisogna inoltre far notare che si ha anche la possibilità di caricare ed eseguire funzioni lambda che fanno parte di un container Docker [9]. Come si vedrà, questa possibilità che offre il servizio AWS Lambda comporterà dei vantaggi importanti per il caricamento e l'esecuzione di codici sorgenti anche molto grandi in dimensione.

2.1.3 API Gateway

Per permettere l'interfacciamento ad una funzione lambda tramite una semplice API nel protocollo Hypertext Transfer Protocol (HTTP), si ha la necessità di utilizzare il servizio API Gateway.

Amazon API Gateway è un servizio completamente gestito che semplifica agli sviluppatori la pubblicazione, la manutenzione, il monitoraggio e la protezione delle API su qualsiasi scala, gestendo tutte le attività di accettazione ed elaborazione relative a centinaia di migliaia di chiamate API simultanee, inclusi gestione del traffico, controllo di accessi e autorizzazioni, monitoraggio e gestione delle versioni delle API. Bastano pochi clic sulla Console di gestione AWS per creare un'API che agisca come porta d'entrata attraverso la quale le applicazioni, quali ad esempio le funzioni lambda, possono accedere a dati, logica di business o funzionalità dei servizi di *backend*.

Il costo del servizio API Gateway, inoltre, dipende soltanto dal numero delle chiamate alla API (che può essere una REST API o una semplice API) e dai dati trasferiti in uscita.

2.1.4 Amazon Simple Queue System

Per il coordinamento tra le funzioni lambda, che sarà descritto in maniera approfondita nel capitolo successivo, si ha la necessità di usare il servizio AWS Simple Queue Server (SQS).

AWS SQS è un sistema di creazione e gestione di code, alle quali è possibile inviare dei messaggi che possono essere successivamente gestiti da servizi e architetture distribuite, quali ad esempio delle funzioni lambda, che tramite un meccanismo di *polling* possono prelevare i messaggi ed elaborarli con successo.

Rispetto a servizi equivalenti messi a disposizione dai competitor (Microsoft Azure, Google Cloud, ...) , Amazon SQS richiede una configurazione minima senza sovraccarico amministrativo. Inoltre, Amazon SQS funziona su vasta scala ed è in grado di elaborare miliardi di messaggi al giorno.

2.1.5 Amazon Cognito

Per la gestione dell'autenticazione e delle autorizzazioni al progetto, si farà uso della piattaforma Amazon Cognito. Seppur questo servizio non verrà trattato in maniera approfondita nelle seguenti sezioni, non facendo parte del focus della tesi, si è deciso comunque di riportare una breve descrizione che ne illustri le caratteristiche.

Amazon Cognito è un servizio che consente di aggiungere strumenti di registrazione e autenticazione ad applicazioni Web e per dispositivi mobili, permettendo di autenticare gli utenti tramite un provider di identità esterno e fornendo credenziali di sicurezza temporanee per accedere alle risorse AWS di *backend* di un'app o qualsiasi servizio protetto da Amazon API Gateway. Amazon Cognito è anche compatibile con provider di identità esterni che supportano lo standard SAML o OpenID Connect, e provider di identità social quali Facebook, Twitter e Amazon, e consente anche di integrare il proprio provider di identità.

La gestione delle autenticazioni in cognito avviene tramite due componenti principali:

- pool di utenti: usati per garantire le funzionalità di accesso agli utenti dell'applicazione che ne fa uso.
- pool di identità: usati per garantire un accesso (eventualmente temporaneo) ad altri servizi AWS per gli utenti che ne fanno uso.

2.1.6 Amazon Virtual Private Cloud

Amazon Virtual Private Cloud (VPC) è un servizio che consente di definire delle reti virtuali personalizzate in cui vengono avviate un insieme di risorse. Questa rete virtuale è simile a una comune rete da gestire all'interno del proprio data center, ma con i vantaggi dell'infrastruttura scalabile di AWS. Tramite un uso corretto del servizio si ha la possibilità di isolare, a livello di rete, alcune delle risorse di un'applicazione, garantendo una maggior sicurezza verso gli attacchi esterni, in quanto si riduce la superficie di attacco.

Per definire una rete tramite il Virtual Private Cloud, è necessaria la definizione e la configurazione di diverse componenti rilevanti, quali ad esempio:

- Sottorete (*subnet*): un segmento di un intervallo di indirizzi IP di cloud privato virtuale (Virtual Private Cloud, VPC) in cui collocare gruppi di risorse isolate. Ciascuna sottorete può essere privata o pubblica: una sottorete pubblica può comunicare con Internet, sia per connessioni in ingresso che in uscita, mentre una sottorete privata generalmente non possiede accessi verso l'esterno, a meno che non si usi una NAT Gateway.
- Tabella di routing: Contiene un insieme di regole denominate *route* che consentono di determinare la direzione del traffico di rete.
- Internet Gateway: un gateway collegato al VPC per consentire la comunicazione tra le risorse del VPC e Internet (connessioni verso l'esterno).
- NAT Gateway: Gateway che consente, alle istanze di una sottorete privata, di connettersi ad Internet o ad altri servizi AWS (traducendo l'indirizzo IP della sottorete privata in un IP pubblico), impedendo allo stesso tempo le comunicazioni in ingresso provenienti da connessioni esterne alla sottorete.

2.1.7 Amazon Relational Database Service

Amazon Relational Database Service (RDS) è un servizio gestito che consente di configurare, utilizzare e ridimensionare le risorse di database relazionali nel cloud in maniera semplice ed efficace. Offre funzionalità ridimensionabili a un costo vantaggioso e gestisce al contempo le lunghe attività amministrative del database, per consentire all'utente di concentrarsi sulle proprie applicazioni e sul business. Tramite RDS è possibile definire e allocare risorse per l'esecuzione di istanze di database quali MySQL, MariaDB, Oracle, SQL Server o PostgreSQL.

Nel contesto del progetto oggetto della presente tesi, si è scelto di utilizzare Amazon RDS per l'istanziamento e l'utilizzo di un database PostgreSQL, utilizzato come database primario per memorizzare le informazioni rilevanti dell'applicazione.

2.1.8 AWS CloudFormation e AWS Cloud Development Kit

Per la definizione e il *deploy* dell'architettura cloud, si è scelto di utilizzare uno strumento che fosse in grado di definire le risorse del progetto in maniera programmatica. Infatti, seppur sia possibile utilizzare la console di AWS per creare nuove risorse in maniera semplice ed intuitiva, al crescere del numero delle risorse del progetto si ha una manutenzione delle risorse che diventa sempre più complessa e difficile da gestire. Utilizzando servizi di AWS quali CloudFormation, tuttavia, si ha la possibilità di definire in maniera dichiarativa le risorse per il progetto, eventualmente interconnesse tra di loro, all'interno di un semplice file in formato YAML o JSON, permettendo la definizione e la creazione di uno *Stack* (una collezione di risorse in CloudFormation) in maniera incrementale (ad ogni modifica che viene fatta, infatti, viene creato un *change set* che implementa esclusivamente le modifiche effettuate rispetto al *deploy* precedente).

Nello specifico del progetto, per la definizione dell'infrastruttura si è deciso di utilizzare la libreria AWS Cloud Development Kit (CDK) [10], un *framework* di sviluppo software *open source* che consente di definire risorse di applicazioni cloud tramite linguaggi di programmazione noti, quali ad esempio TypeScript

o Python. Questo *framework* consente di definire in maniera programmatica le risorse che devono essere inserite nell'architettura cloud dell'applicazione, e dopo aver definito l'architettura CDK si occupa di sintetizzare un file in formato YAML, che definisce le risorse che verranno istanziate all'interno di uno *Stack* CloudFormation, consentendo una gestione efficace e scalabile delle risorse.

2.2 PostgreSQL

Dopo aver descritto la scelta della piattaforma di *cloud computing*, così come i servizi utilizzati, si può quindi ora parlare delle altre risorse software usate per la progettazione e lo sviluppo del progetto.

Per quanto riguarda il database da utilizzare per gestire i dati del progetto, si è deciso di utilizzare PostgreSQL. Alcuni dei motivi che hanno guidato la scelta di tale database verranno illustrati in fase di progettazione. In questa sezione ci si limiterà a descrivere PostgreSQL come database, illustrando alcune delle funzionalità che offre.

PostgreSQL [11] è un *database management system* relazionale (RDBMS) *open source*, rilasciato inizialmente nel 1996. Nel tempo tale database, rispetto ai concorrenti, ha goduto di un successo sempre maggiore, diventando di fatto uno dei database relazionali più affidabili e con più funzionalità presenti sul mercato.

Alcune delle *feature* supportate sono, ad esempio:

- possibilità di definire dei tipi da parte dell'utente;
- meccanismo di ereditarietà tra le tabelle di un database;
- meccanismo di *locking* per la gestione della concorrenza sofisticato, ed eventualmente supporto per il Multi-version Concurrency Control (MVCC);
- s per le transazioni innestate;
- supporto alla replicazione asincrona.

Oltre a queste funzionalità si ha anche un ottimo supporto, da parte del database, ai campi di natura semistrutturata, espressi in formato jsonb (formato

json memorizzato in un formato binario per una maggior efficienza computazionale, a causa di un *overhead* minore nel *parsing* del documento), permettendo la possibilità di combinare, alla struttura relazionale tradizionale, anche la struttura semistrutturata che caratterizza i database NoSQL, garantendo una maggiore flessibilità nella gestione e nella rappresentazione dei dati. Questa *feature*, come si vedrà nel capitolo successivo, è risultata particolarmente importante per la progettazione del database del progetto Parola di Motoneurone.

2.3 GraphQL

Per quanto riguarda la realizzazione del *backend* del progetto, effettuata in un contesto *serverless* tramite la definizione di funzioni lambda su AWS, si è deciso di utilizzare GraphQL [12] per la realizzazione della API del servizio, consentendo una corretta comunicazione tra il *client* ed il *backend*.

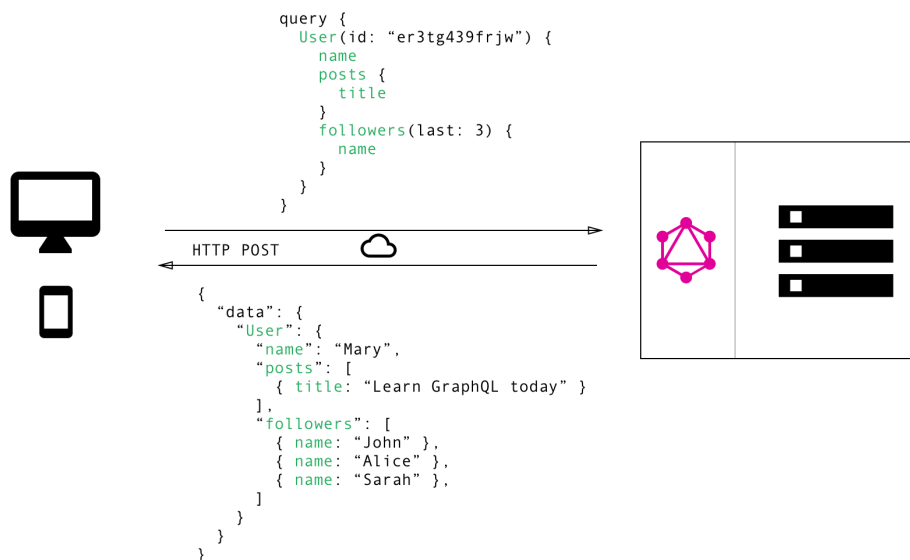


Figura 2.1: Esempio di query nel linguaggio di interrogazione GraphQL, in cui il client specifica nello specifico le informazioni richieste

GraphQL è un linguaggio di interrogazione e manipolazione dei dati *open source* per API, provvisto di un *runtime* per soddisfare *query* con dati esistenti. GraphQL fu sviluppato internamente a Facebook nel 2012, prima di esser reso

pubblico nel 2015. Ciò che contraddistingue GraphQL dagli altri standard per la realizzazione di API web (REST, SOAP, ...) è il fatto che si ha un unico *endpoint* per la comunicazione con il server, in cui il *client* specifica esattamente ciò di cui ha bisogno per il corretto funzionamento. Come è possibile notare dalla Figura 2.1, al contrario di una REST API, l'uso di GraphQL evita il problema dell'*overfetching*, in cui il client, ricevendo risposte in un formato fisso, può ricevere più dati di quanto richiesto, e dell'*underfetching*, in cui il *client* necessita di contattare più *endpoint* del previsto per avere le informazioni di cui necessita. La possibilità di usare un solo *endpoint*, inoltre, si adatta bene in un contesto *serverless*, in quanto permette di definire un'unica funzione lambda che soddisfi la richiesta, senza creare necessariamente una funzione per ogni possibile rotta di una REST API. Si può infine notare che in GraphQL si ha la necessità di definire degli schemi espliciti per ciascuno dei tipi di dato coinvolti nelle *query*, fornendo agli sviluppatori del *client* che vi si interfaccia tutta la documentazione necessaria per il corretto interfacciamento.

Per l'implementazione di un *backend* che faccia uso di GraphQL, si è quindi utilizzato Apollo Server [13]. Apollo Server è una libreria *open source*, realizzata in NodeJS, che consente di definire ed implementare un servizio di *backend* performante e compatibile con le specifiche di GraphQL.

2.4 Ambienti e linguaggi di programmazione usati

Gli ambienti e i linguaggi di programmazione usati per il progetto sono:

- **NodeJS [14] e Typescript [15]:** usati per la definizione e il *deploy* dell'architettura cloud, tramite l'uso di AWS CDK, e per la realizzazione del *backend*, utilizzando Apollo Server.
- **Python [16]:** usato per l'implementazione, nel progetto, di diverse funzioni lambda, al fine di renderle compatibili con gli algoritmi di analisi multimediale che dovranno essere in seguito implementati all'interno del progetto.

Capitolo 3

Parola di motoneurone: progettazione dell'architettura cloud e del database

In questo capitolo verrà descritta la fase di progettazione della parte di *backend* e architetturale relativa al progetto Parola di Motoneurone. Si partirà da una descrizione più specifica del progetto e delle applicazioni che sono coinvolte nella piattaforma, che saranno poi usate per definire i requisiti funzionali e non funzionali da soddisfare. Si passerà quindi a descrivere nel dettaglio la fase di progettazione relativa all'architettura cloud del sistema, al *backend* e al database.

3.1 Descrizione progetto

Come precedentemente descritto nella sezione 1.3, l'obiettivo del progetto Parola di Motoneurone è quello di realizzare un sistema per il monitoraggio da remoto di un paziente, affetto da una patologia neurologica. Questo monitoraggio avviene tramite una valutazione della disartria proveniente da dati multimediali, di natura audio e video, ottenuti da un'applicazione mobile, usata dal paziente per riprendersi mentre svolge degli esercizi.

Tale sistema è quindi composto da tre componenti principali:

- l'applicazione mobile dedicata al paziente per l'acquisizione dei dati, in cui il soggetto si riprenderà mentre esegue degli esercizi, i cui contenuti verranno poi caricati all'interno di uno *storage* apposito per l'elaborazione successiva;
- l'applicazione web del clinico, che visionerà gli indici relativi agli esercizi svolti dal paziente nelle varie sessioni, calcolati automaticamente da appositi algoritmi di analisi multimediale;
- la componente architetturale e di *backend* del sistema, che si occuperà di gestire la comunicazione tra l'applicazione lato paziente e l'applicazione lato clinico, garantendo l'interfacciamento con una base di dati condivisa. Questa componente si occuperà inoltre di gestire ed eseguire i vari algoritmi di analisi audio/video in maniera automatica e scalabile.

In questa tesi si parlerà, in particolare, della realizzazione di quest'ultima componente. Per definire i requisiti funzionali e non funzionali che dovranno essere soddisfatti, tuttavia, sarà necessario innanzitutto descrivere più nel dettaglio le applicazioni del paziente e del clinico, evidenziando i casi d'uso e i *workflow* associati.

3.1.1 Specifiche dell'applicazione del paziente

In questa sezione è descritto il flusso di utilizzo dell'applicazione del paziente, dettagliando i casi d'uso che devono essere soddisfatti.

Il paziente, al primo accesso dell'applicazione, deve registrarsi tramite un invito dal clinico che lo segue, che specificherà per il paziente gli esercizi che deve svolgere. Dopo aver inserito i dati di rilievo per la registrazione dell'account, il paziente potrà fare uso dell'applicazione.

Dopo il login, il paziente potrà quindi visionare le sessioni che dovrà effettuare, con l'opzione di iniziarne una nuova. Iniziando una sessione, al paziente saranno visualizzati sequenzialmente gli N esercizi da svolgere, contrassegnati ciascuno da un titolo, una descrizione, e da una durata massima o predefinita. Il paziente,

quindi, si riprenderà mentre svolge l'esercizio, che verrà fatto partire con un *countdown*, eventualmente con la possibilità di svolgerlo da capo nel caso di una esecuzione non corretta. Al termine di ogni esercizio, quindi, verrà memorizzato un file multimediale (audio o video) relativo all'esercizio.

Dopo aver svolto tutti gli esercizi della sessione tutti i dati multimediali saranno compressi in un unico file in formato zip, che sarà successivamente inviato al servizio di *storage* per la memorizzazione e la successiva elaborazione da parte degli algoritmi di analisi. Il file così memorizzato potrà eventualmente anche essere visionato dal clinico, che vuole monitorare gli esercizi svolti dal paziente. Nel caso in cui il caricamento del file avviene con successo, la sessione è completata e lo .zip viene rimosso dallo *storage* in locale. Nel caso in cui, invece, il caricamento non andasse a buon fine, l'applicazione inserirà il file da caricare in una coda, e quando possibile procederà di nuovo con il suo *upload*.

Un diagramma UML che rappresenta sinteticamente gli use case descritti è presente nella Figura 3.1. Oltre al funzionamento dell'applicazione e ai suoi casi d'uso, sono state definite in accordo con la componente clinica degli Ospedali Riuniti di Ancona alcune metriche utili in fase di progettazione. Tali metriche sono:

- Il numero di pazienti di un reparto che usano l'applicazione nel corso di una settimana, pari a 30.
- La frequenza media delle sessioni che paziente dovrà svolgere, pari a circa 2 volte alla settimana.
- La dimensione media del file di una sessione inviato dall'applicazione del paziente, che è stato stimato pari a circa 70 MB (stima ottenuta tramite lo sviluppo di un prototipo dell'applicazione per effettuare le acquisizioni). I formati considerati per il salvataggio dei dati video e audio sono, rispettivamente, WebM e MP3.
- La velocità minima di *upload* relativa alla connessione di cui fa uso il paziente, supposta pari a 1 Mbps. In queste condizioni, si avrebbe un tempo

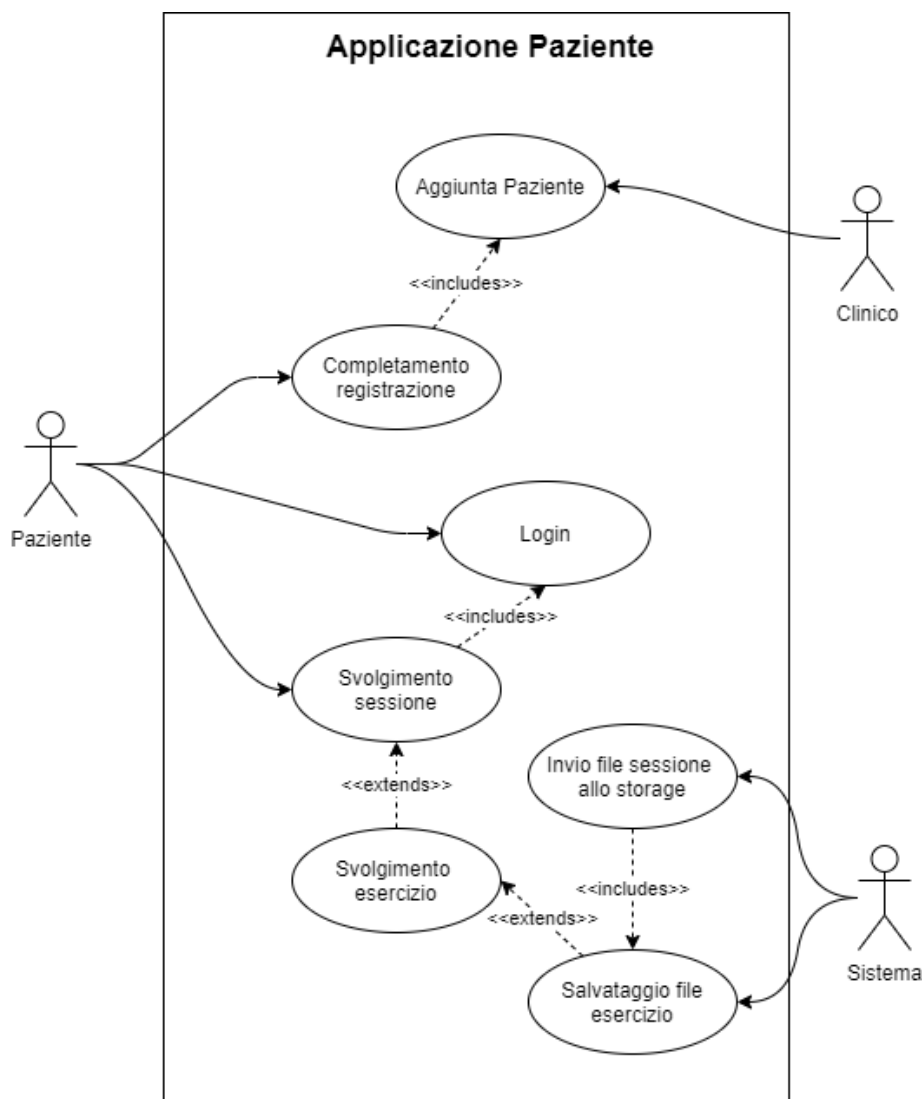


Figura 3.1: Use case dell'applicazione del paziente

di caricamento del file (supposto, sempre, di dimensioni pari a 70MB) pari a circa 10 minuti. Questo è il tempo di *upload* considerato nel caso peggiore.

3.1.2 Specifiche dell'applicazione del clinico

Per quanto riguarda il *workflow* e i casi d'uso relativi all'applicazione del clinico, bisogna innanzitutto chiarire il suo ruolo all'interno del sistema: ogni clinico lavora in un ospedale specifico, e fa riferimento ad un unico reparto relativo all'ospedale stesso. Nello specifico dell'applicazione, è possibile quindi distinguere

due tipi di clinici: il **caporeparto**, che è il responsabile per l'uso della piattaforma in un reparto specifico, avendo la possibilità di aggiungere degli account relativi a clinici che afferiscono a quel reparto, ed il **clinico** tradizionale, afferente a un singolo reparto, che segue il paziente e monitora i suoi progressi.

Nello specifico dell'applicazione del clinico, quindi, è possibile individuare le seguenti funzionalità principali:

- Aggiunta di un nuovo clinico: un caporeparto può decidere di aggiungere, modificare, aggiornare, ed eliminare i dati relativi ad un clinico della piattaforma, creando od eliminando eventualmente un account con cui costui può accedere.
- Aggiunta di un nuovo paziente, che può essere effettuata da un caporeparto o dal clinico che segue il paziente. Al paziente, in particolare, può essere assegnato più di un clinico che lo segue. Dopo l'aggiunta del paziente, verrà quindi inviata una mail che consentirà la creazione dell'account per l'applicazione del paziente.
- La modifica e l'eliminazione di un paziente, che possono essere effettuate esclusivamente dal caporeparto. Bisogna tuttavia notare che l'eliminazione di un paziente da un reparto non comporta necessariamente la sua eliminazione dal database, in quanto un paziente può essere associato, eventualmente, a più reparti.
- La visualizzazione dei pazienti, che può essere effettuata dal caporeparto (che ha la possibilità di vedere tutti i pazienti associati al reparto) o da uno dei clinici del reparto (che può vedere esclusivamente i pazienti che segue). Bisogna notare che verrà utilizzato, in questo caso, un sistema di filtraggio dei pazienti tramite l'uso di *tag*, che sono delle stringhe usate come etichette per catalogare un paziente e che possono essere filtrate dal clinico. In particolare ad ogni paziente di un reparto viene associato un insieme di *tag* (decise a livello di reparto), permettendo quindi un filtraggio efficace

incrociando, eventualmente, *tag* differenti. È prevista, inoltre, la possibilità di ricercare il paziente per nome e per cognome.

- La visualizzazione del singolo paziente, in cui si ha la possibilità di visualizzare i dettagli anagrafici, di conoscere i dettagli di ciascuna delle sessioni effettuate dal paziente e, infine, di visualizzare gli indici associati agli esercizi effettuati dal paziente, ordinati per il tempo d'esecuzione dell'esercizio.

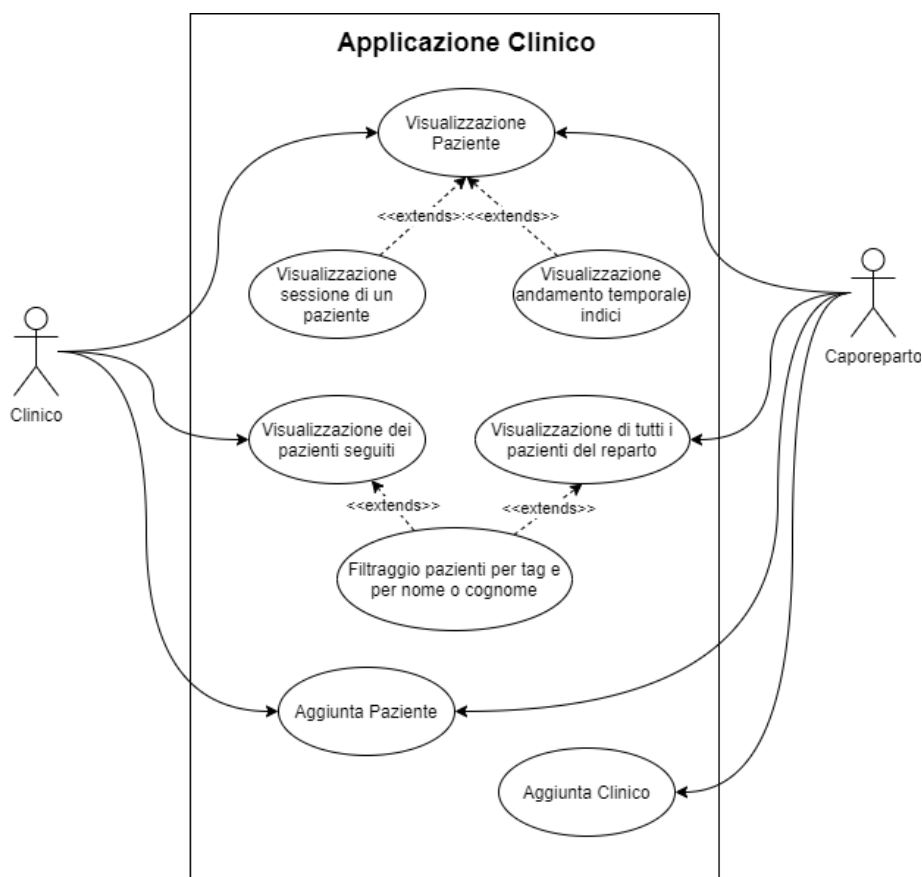


Figura 3.2: Use case dell'applicazione del clinico

Un diagramma UML che rappresenta i casi d'uso dell'applicazione è presente in Figura 3.2. Anche in questo caso sono state individuate alcune metriche di rilievo, utili in fase di progettazione. Tali metriche sono:

- Il numero dei clinici: si suppone, in media, che per ogni reparto siano presenti 1 caporeparto e 3 altri clinici del reparto. Questo numero potrebbe crescere o diminuire in base alle dimensioni del reparto.
- Il numero di nuovi pazienti seguiti dal clinico nell'arco temporale di 1 anno, pari a 100. La frequenza con cui si presentano nuovi pazienti nel corso dell'anno, inoltre, può essere considerata omogenea: non ci sono periodi dell'anno in cui si presentano più pazienti.
- Il numero di consultazione degli indici che viene effettuata da un clinico per paziente, che è stata considerata pari a 10 alla settimana. Questo numero, evidentemente, può variare sulla base della patologia, del paziente o del reparto considerati.

3.2 Analisi dei requisiti

Avendo definito le specifiche relative all'applicazione per il paziente e all'applicazione per il clinico, è quindi ora possibile descrivere qual è il ruolo della componente del *backend* all'interno del sistema, delineando i requisiti, funzionali e non funzionali, che devono essere rispettati.

3.2.1 Requisiti funzionali

Per garantire un corretto interfacciamento tra le due applicazioni e un corretto funzionamento della piattaforma, l'infrastruttura cloud che dovrà essere implementata deve rispettare i seguenti requisiti funzionali:

- Il sistema deve gestire i file .zip degli esercizi caricati dai pazienti, salvandoli in un apposito *storage* e garantendo caratteristiche quali persistenza, integrità, ed eventualmente anche la possibilità di cifrare i dati. I file all'interno dello *storage* devono essere consultabili da parte dei clinici (per verificare, per l'esempio, l'andamento degli esercizi di una sessione).
- Il sistema deve esporre un'interfaccia che permetta, all'applicazione del paziente, di caricare il file compresso contenente i dati audio e video relativi

agli esercizi di una sessione del paziente, che dovranno poi essere elaborati tramite appositi algoritmi di intelligenza artificiale e *deep learning* per la successiva analisi.

- Il sistema deve essere in grado di distribuire, correttamente ed automaticamente, i file multimediali relativi ad una sessione tra i diversi algoritmi di analisi.
- Il sistema deve consentire il processamento automatico di ogni file multimediale, relativo ad un esercizio svolto dal cliente, da parte dell'algoritmo di analisi associato, al fine di ottenere un indice, che costituisce una valutazione oggettiva dell'esercizio e che sarà quindi caricato all'interno del database del sistema, per consentire la successiva consultazione da parte del clinico.
- Il sistema per la distribuzione ed il processamento di un file deve essere facilmente estensibile consentendo l'aggiunta di nuovi esercizi ed algoritmi per l'analisi, per far fronte all'eventuale possibilità di estendersi ad altre malattie neurologiche o di includere nuove tipologie di esercizi e valutazioni.
- Il sistema dovrà presentare un database che sia in grado di memorizzare le informazioni di interesse per il corretto funzionamento di entrambe le applicazioni, e di gestire le richieste e le transazioni provenienti dai *client*.
- Il sistema dovrà presentare un *backend* per consentire una corretta comunicazione con le due applicazioni, fornendo funzionalità di Create, Read, Update e Delete (CRUD) sui dati rilevanti alle applicazioni.
- Il sistema deve gestire correttamente la registrazione e il login sia del paziente che dei clinici (eventualmente caporeparti).

Per quanto riguarda la componente di *backend*, che dovrà essere implementata all'interno del sistema, le funzionalità richieste sono:

- CRUD dei pazienti;

- CRUD dei clinici;
- CRUD degli ospedali e dei reparti associati;
- CRUD delle tag per ciascun reparto;
- CRUD degli account associati ai pazienti ed ai clinici;
- CRUD di una sessione di un paziente;
- CRUD degli esercizi che possono essere assegnati ad un paziente con le informazioni sui relativi indici di performance, consentendo inoltre l'inserimento dei risultati delle sessioni, ottenuti dagli algoritmi di analisi.

3.2.2 Requisiti non funzionali

I requisiti non funzionali relativi alla progettazione e all'implementazione della componente architetturale e di *backend* dell'applicazione sono i seguenti:

- Il sistema deve essere implementato tramite tecnologie e servizi offerti da Amazon Web Services (AWS).
- Il sistema deve essere scalabile, riuscendo a gestire richieste e carichi di dati anche notevolmente superiori a quelli previsti per il funzionamento.
- L'interfaccia esposta dalla API del *backend* deve essere chiara e semplice da usare dal *client*, per permettere un interfacciamento veloce e facilmente integrabile.
- Il codice usato per lo sviluppo del progetto deve essere facilmente mantenibile ed estensibile garantendo la possibilità di adottare un approccio agile, in cui le funzionalità verranno introdotte incrementalmente tramite approcci di *continuous deployment* e *continuous integration*.
- L'architettura cloud deve essere definita in modo tale da essere semplice da monitorare e mantenere, accomodando in maniera flessibile eventuali modifiche introdotte in seguito al *deploy*.

3.3 Progettazione dell'architettura cloud

Definiti i requisiti funzionali e non funzionali da soddisfare, è ora possibile parlare nello specifico della fase di progettazione relativa all'architettura cloud da realizzare. L'approccio adottato nella progettazione di questa architettura è di tipo *bottom-up*, in cui si è partiti dai requisiti specifici da soddisfare, effettuando in seguito l'integrazione dei vari componenti fino ad avere lo schema finale dell'architettura da realizzare.

3.3.1 Scelta del servizio di storage per il salvataggio degli esercizi

Per soddisfare il requisito di memorizzazione dei dati relativi alle sessioni di un paziente si è deciso di utilizzare il servizio Amazon S3. In particolare, è stato definito un *bucket* apposito per il caricamento dei file associati a ciascun paziente. All'interno del *bucket* i file vengono disposti all'interno di diverse cartelle, che identificano il paziente di riferimento.

3.3.2 Interfacciamento dell'applicazione del paziente con lo storage

Per quanto riguarda l'interfacciamento del *bucket* appena introdotto con l'applicazione del paziente per il caricamento del file di una sessione, era stato deciso in prima istanza di usare il servizio API Gateway per consentire l'*upload* del file. Tale scelta tuttavia era inadatta, a causa del limite massimo di caricamento imposto da API Gateway pari a 10MB, che è molto minore della dimensione media di un file compresso di una sessione, supposto in media di dimensioni pari a 70MB.

Per risolvere il problema, è stata utilizzata una funzione lambda che consente la generazione di un *URL presigned* [17]. Gli *URL presigned* sono uno strumento tipicamente usato per garantire l'accesso a un *bucket* S3 per un'applicazione esterna tramite la condivisione di un link, valido per un certo periodo temporale, generato da un servizio che ha i permessi necessari per l'accesso al *bucket*. Questo

CAPITOLO 3. PAROLA DI MOTONEURONE: PROGETTAZIONE DELL'ARCHITETTURA CLOUD E DEL DATABASE

link permette l'accesso diretto al *bucket* per il caricamento o la lettura dei file, e non possiede limiti di alcun tipo relativamente alla dimensione massima del file da caricare. In questo caso, in particolare, la funzione lambda genererà un link, inviato al *client*, che consente il caricamento di un file all'interno del *bucket*, a cui sarà associato un particolare nome. Questo link non ha limiti per quanto riguarda la dimensione massima di caricamento, superando la problematica derivante dall'utilizzo dell'API Gateway.

Integrando il sistema sopra descritto con l'applicazione associata al paziente e il resto dell'architettura, è stato ottenuto il *workflow* descritto dalla Figura 3.3

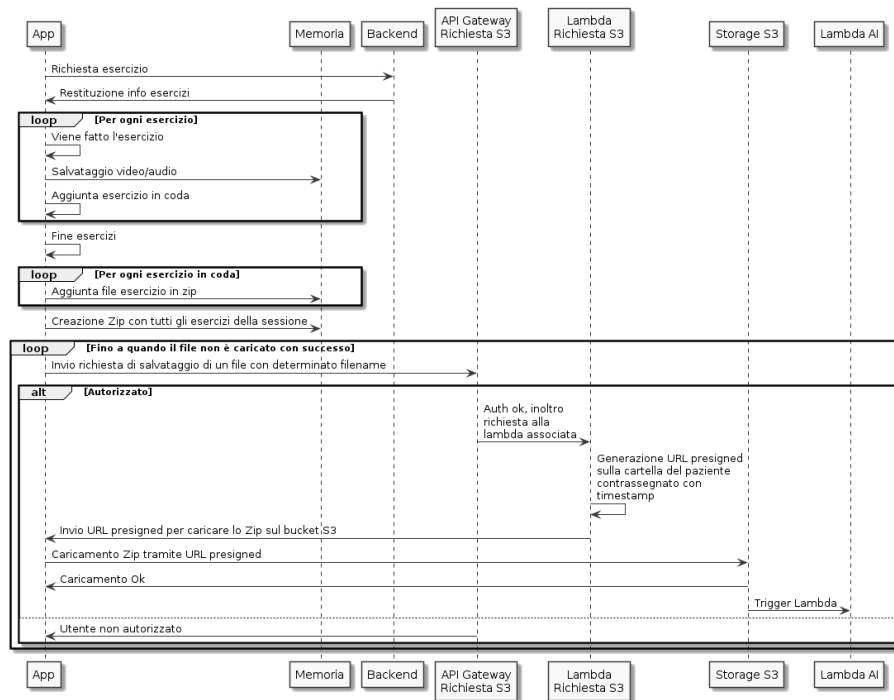


Figura 3.3: Sequence diagram che illustra lo svolgimento degli esercizi all'interno dell'applicazione del paziente, con il conseguente caricamento all'interno dello storage

3.3.3 Scelta della componente per l'esecuzione degli algoritmi di analisi

Definito il servizio da usare per lo *storage* dei file associati alle sessioni del paziente, il passo successivo è stato quello di identificare il servizio da utilizzare per l'esecuzione degli algoritmi di analisi, che fanno uso di tecniche di *machine learning* e *deep learning* per ottenere un indice a partire da un file multimediale.

A tal proposito, è stata effettuata una breve sperimentazione al fine di valutare se usare il servizio EC2 di AWS, che consente di implementare delle macchine virtuali scalabili, o il servizio AWS Lambda, per l'esecuzione degli algoritmi di analisi in un contesto *serverless*. Per l'esecuzione dell'esperimento è stata considerata una rete resnet50 implementata tramite la libreria pytorch [18], per valutare i tempi d'esecuzione e i costi nei due servizi.

Per quanto riguarda il servizio EC2, è stato effettuato un test su una macchina virtuale di tipo t2.large, che mette a disposizione 2 vCPU e 8GB di RAM per l'elaborazione. In un'istanza di questo tipo, è stato accertato un tempo medio di elaborazione tra i 500ms e i 600ms per l'esecuzione della rete su un'immagine prelevata da internet. Il costo che deriverebbe dall'uso di tale istanza sarebbe pari a 66.82 dollari al mese, che è un costo fisso e indipendente dalla frequenza con cui la rete viene invocata.

Per quanto riguarda l'uso di una funzione in AWS Lambda, bisogna evidenziare una problematica emersa nell'esecuzione della rete, derivante dal fatto che le dimensioni di librerie quali pytorch sono eccessivamente elevate nel caso del caricamento diretto del codice all'interno della funzione lambda. Per risolvere il problema è possibile fare uso di un sistema che, al primo avvio della funzione, decomprime i contenuti di un file compresso, contenente la libreria, all'interno della cartella `/tmp`, al fine di consentire il suo utilizzo durante l'esecuzione della funzione, essendo la massima dimensione consentita di questa cartella pari a 512MB. Questa soluzione, tuttavia, comporta dei tempi di primo avvio (*cold boot*) molto elevati, a causa della necessità di decomprimere grandi quantità di file. Una soluzione più efficace, che è stata usata nel contesto di questa analisi, consiste invece

nell'utilizzo di un container Docker, che viene eseguito dalla funzione lambda. Questa funzionalità, abbastanza recente, consente l'uso di container di dimensioni fino a 10GB, che sono sufficienti per l'esecuzione degli algoritmi di analisi considerati nel contesto del progetto.

Usando una funzione lambda tramite container, quindi, è stato ottenuto un tempo medio di elaborazione pari a circa 600ms nel caso di una funzione con 2GB di memoria già invocata recentemente, e pari a circa 10 secondi nel caso di una funzione lambda che non è stata invocata di recente (*cold boot*). Per quanto riguarda i costi, invece, è possibile notare un drastico calo nel costo mensile da sostenere, soprattutto considerando i dati forniti nella sezione 3.1.1 per il calcolo del numero delle richieste. La presenza di costi così poco pronunciati può essere spiegata dal fatto che le invocazioni degli algoritmi di analisi nel caso del progetto Parola di Motoneurone non sono molto frequenti e non avvengono in maniera continuata, e il costo delle funzioni lambda, al contrario delle istanze EC2, dipende esclusivamente dal numero di invocazioni e dal tempo totale di computazione. In particolare, considerando 30 pazienti che effettuano 30 esercizi nel corso di una sessione, e considerando lo svolgimento di 2 sessioni alla settimana (dati di esempio concordati con i partner clinici degli Ospedali Riuniti di Ancona), si avrebbe un numero totale di invocazioni mensili pari a 7200, e un tempo di esecuzione mensile pari a 11760 secondi (considerato, per ogni sessione, un tempo di esecuzione pari a 49 secondi, in cui 20 secondi sono dati dal primo avvio e i restanti 29 secondi sono dati dall'esecuzione degli altri esercizi dopo il primo avvio, con una durata supposta pari a 1 secondo), a cui corrisponderebbe un costo totale per l'esecuzione pari a circa 0.23 dollari.

Dall'analisi appena descritta è quindi possibile dedurre come, a parità di prestazioni, il costo nell'uso delle funzioni lambda è molto minore rispetto all'uso di un'istanza EC2. Questo motivo, assieme alla maggior facilità nella gestione e nella scalabilità di una funzione lambda in base alle richieste, ha motivato la scelta di questo servizio per la computazione degli algoritmi di analisi, che eventualmente invocheranno dei container Docker per evitare il problema della dimensione massima del codice descritto in precedenza.

3.3.4 Distribuzione automatica degli esercizi per ciascun algoritmo

Avendo deciso di usare delle funzioni lambda per l'implementazione degli algoritmi di analisi, è necessario ora vedere come effettuare la distribuzione automatica dei file multimediali degli esercizi per ciascuno degli algoritmi che li devono elaborare.

La soluzione individuata si basa sull'utilizzo di code definite tramite il servizio Amazon Simple Queue System (SQS), e sulla definizione di una struttura composta da una funzione lambda definita come orchestratore, e da un'insieme di funzioni lambda definite come consumatori, collegate alle code. Utilizzando un sistema di code, associate alle funzioni lambda, si ha la possibilità di garantire una buona scalabilità al sistema: le risorse istanziate per il funzionamento della funzione lambda, infatti, sono proporzionali ai messaggi rimasti sulla coda ancora da elaborare [19]. Questa caratteristica è particolarmente rilevante nel contesto di elaborazioni tramite algoritmi di *deep learning*, in quanto generalmente tali elaborazioni non sono immediate per il singolo file, ed è quindi necessario un sistema che sia in grado di offrire una buona scalabilità all'aumentare della quantità di file da elaborare.

Un diagramma che rappresenta la soluzione è presente in Figura 3.4

In questa soluzione, i consumatori vanno ad implementare i vari algoritmi di analisi che verranno applicati su uno o più file multimediali associati agli esercizi di una sessione, al fine di ottenere degli indici oggettivi che ne indicano la performance. Ognuna di queste funzioni lambda è collegata ad una coda SQS, usata per il raccoglimento di messaggi che indicano la posizione dei file multimediali da elaborare all'interno di un apposito *bucket*, usato come *storage* temporaneo per l'elaborazione dei file. In particolare, ogni consumatore effettua il *polling* della coda associata per ricevere un dato numero di messaggi (batch), a cui verrà associata l'elaborazione del file. In base al numero di messaggi inviati nella coda, la funzione lambda può scalare in maniera automatica. Dopo il calcolo dell'indice effettuato in seguito all'applicazione dell'algoritmo, il consumatore salverà

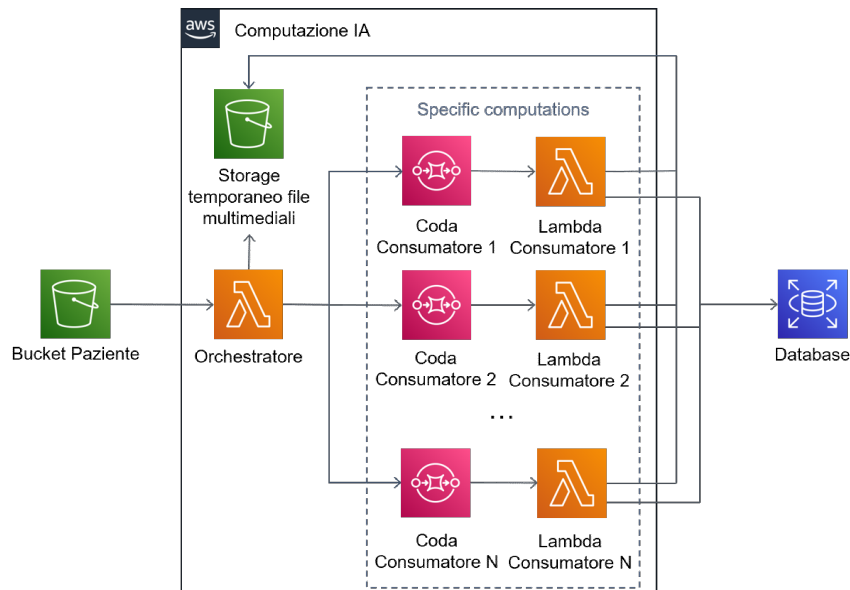


Figura 3.4: Diagramma che rappresenta l'architettura adottata per la distribuzione e la computazione dei file multimediali tramite algoritmi di analisi basati su tecniche di intelligenza artificiale

il risultato così ottenuto all'interno del database del sistema, permettendone la consultazione in un secondo momento da parte del clinico. Dopo l'elaborazione, il file verrà eliminato dal *bucket* temporaneo in cui era situato.

L'orchestratore, invece, è la funzione lambda che si occupa di compiere effettivamente la distribuzione dei file multimediali a uno o più consumatori. L'orchestratore si mette in ascolto sul *bucket* S3 del paziente, al fine di individuare il caricamento di nuovi file, associati a delle sessioni, al suo interno. Quando quindi viene caricato, da parte dell'applicazione del paziente, un file compresso nel formato .zip contenente i dati multimediali degli esercizi di una sessione, l'orchestratore si occupa di:

- Inserire nel database una nuova sessione, con *timestamp* (nel formato ISO 8601) pari all'orario di riceitura del dato;
- Decomprimere il file ricevuto, estraendone i file all'interno del *bucket* per il salvataggio temporaneo precedentemente descritto, usato dai consumatori per effettuare l'analisi sul singolo file audio/video;

CAPITOLO 3. PAROLA DI MOTONEURONE: PROGETTAZIONE DELL'ARCHITETTURA CLOUD E DEL DATABASE

- Individuare gli indici da calcolare in base all'esercizio a cui è assegnato il file, e di conseguenza individuare i consumatori che dovranno elaborare il dato;
- Inviare dei messaggi a ciascuna delle code associate ai consumatori che dovranno elaborare il file.

Una *sequence diagram* che rappresenta il procedimento di distribuzione della computazione degli algoritmi è visibile nella Figura 3.5.

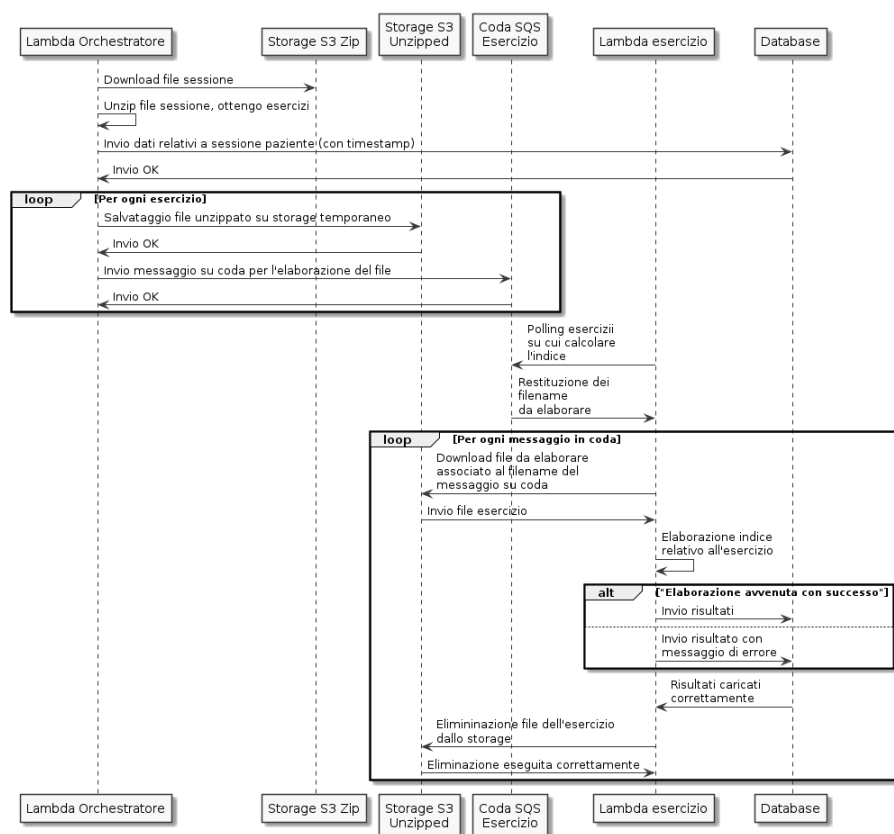


Figura 3.5: Sequence diagram che illustra il procedimento di distribuzione e computazione relativo a una sessione del paziente

Questa struttura, oltre a consentire una computazione automatica e scalabile, ha il vantaggio di essere facilmente estensibile: per aggiungere un nuovo algoritmo di analisi al sistema, infatti, è sufficiente aggiungere un nuovo consumatore e la

coda associata, e modificare il database per aggiungere il nuovo indice, legandolo alla funzione lambda che lo elaborerà.

3.3.5 Gestione dell'autenticazione del paziente e del clinico

Per quanto riguarda la gestione dell'autenticazione del paziente e del clinico, necessario per gestire correttamente le autorizzazioni relative al sistema, si è deciso di fare uso del sistema Amazon Cognito, definendo dei pool di utenti specifici per quanto riguarda i pazienti e i clinici/caporeparto.

In questo contesto, le *sequence diagram* 3.6 e 3.7 presentano i flussi relativi al login e alla registrazione all'interno dell'applicazione del paziente. Per quanto riguarda la registrazione, come è possibile vedere, è possibile utilizzare delle funzioni lambda di pre-registrazione e di post-conferma che consentono, rispettivamente, di controllare la validità del processo di registrazione del paziente e di completare la registrazione associata al paziente, inserendo i dati all'interno del database di sistema.

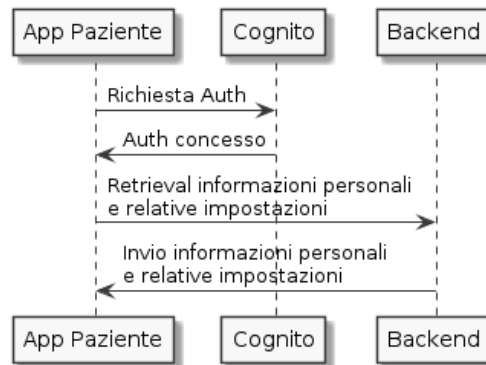


Figura 3.6: Sequence diagram login paziente

Bisogna tuttavia far notare che, nel contesto della presente tesi, l'implementazione della componente di autenticazione non rientra tra gli obiettivi proposti, per cui è stato deciso in prima fase di tralasciare l'implementazione di questo sistema, che verrà riservato per eventuali sviluppi futuri.

CAPITOLO 3. PAROLA DI MOTONEURONE: PROGETTAZIONE DELL'ARCHITETTURA CLOUD E DEL DATABASE

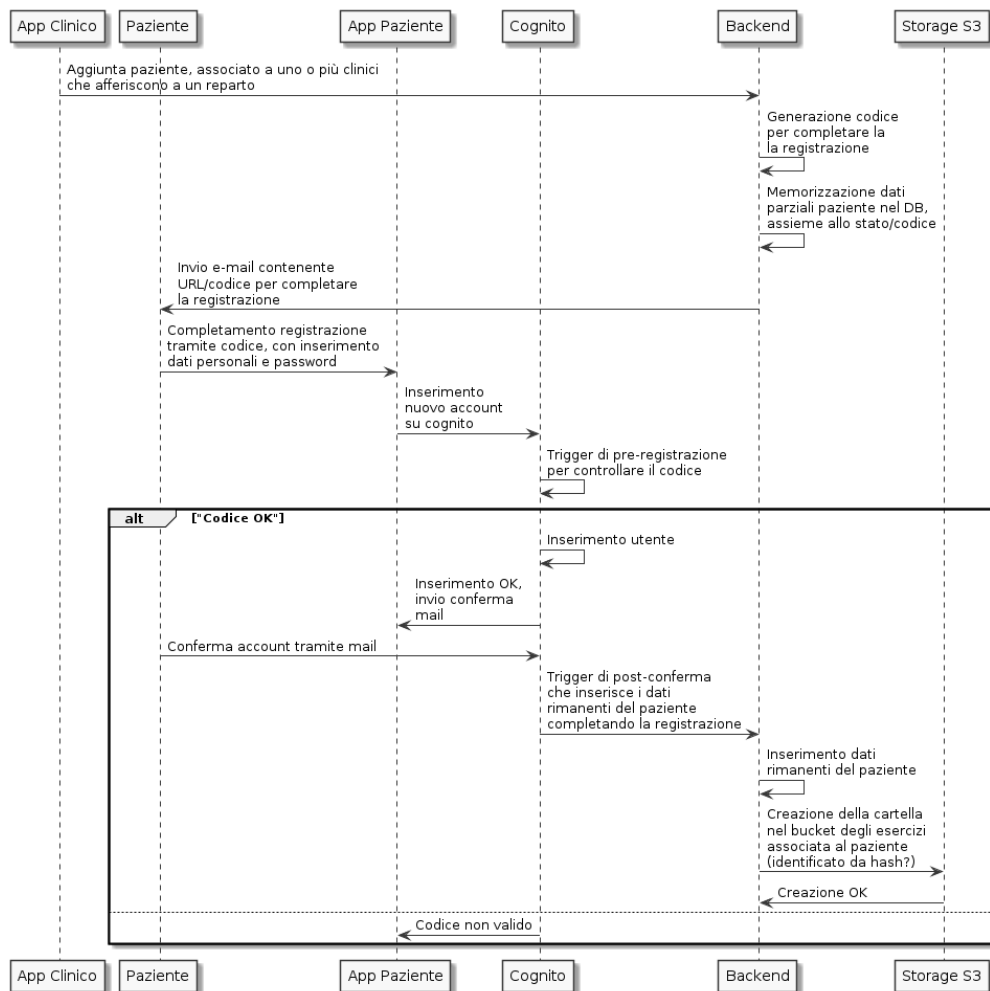


Figura 3.7: Sequence diagram registrazione paziente

3.3.6 Architettura cloud

L'architettura cloud complessiva, ottenuta integrando tutte le osservazioni e le scelte progettuali effettuate nelle sezioni precedenti, è presente in Figura 3.8.

Come è possibile notare dalla Figura, l'architettura è costituita da diversi moduli principali:

- I client del paziente e del clinico, che si interfacciano tramite Amazon Cognito per l'autenticazione, e con il *bucket* S3 contenente i dati dei pazienti.
- Il modulo per consentire l'*upload* dei file delle sessioni all'interno del *bucket* dei pazienti tramite l'uso di *URL presigned*, composto da una funzione

CAPITOLO 3. PAROLA DI MOTONEURONE: PROGETTAZIONE DELL'ARCHITETTURA CLOUD E DEL DATABASE

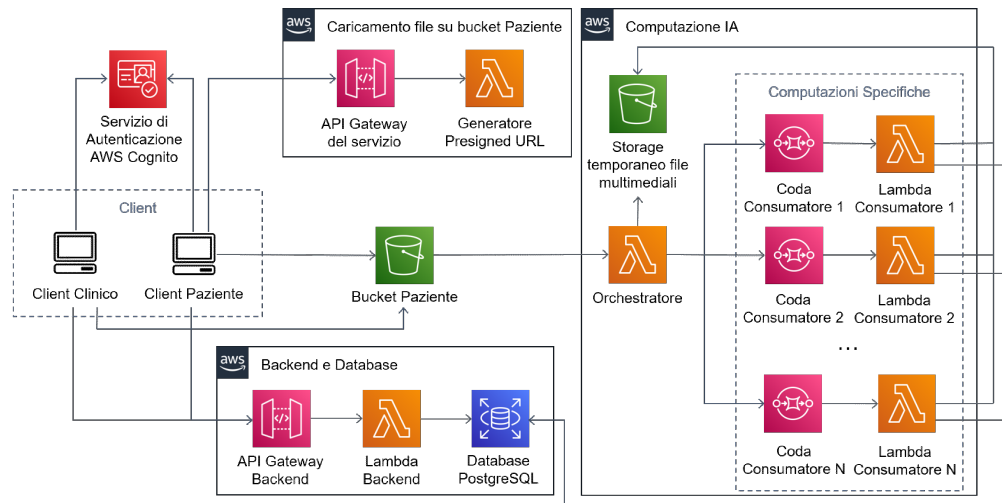


Figura 3.8: Diagramma architetturale dell'applicazione

lambda con relativa API Gateway.

- Il modulo per la computazione degli algoritmi di intelligenza artificiale, che presenta la stessa struttura descritta nella sezione 3.3.4.
- Il modulo che implementa il *backend* del server, che sarà implementata in un contesto *serverless* interfacciandosi ad un database relazionale, la cui progettazione verrà descritta nella sezione successiva.

3.4 Scelta del database e progettazione del modello di dati

In questa sezione sarà trattata la progettazione e la scelta del database per il progetto Parola di Motoneurone.

3.4.1 Progettazione concettuale

Il primo passo della progettazione è consistito, definite le specifiche delle due applicazioni, nel realizzare uno schema concettuale per rappresentare le informazioni di interesse per la piattaforma. Questa progettazione è culminata nella realizzazione dello schema E-R presente in Figura 3.9.

CAPITOLO 3. PAROLA DI MOTONEURONE: PROGETTAZIONE DELL'ARCHITETTURA CLOUD E DEL DATABASE

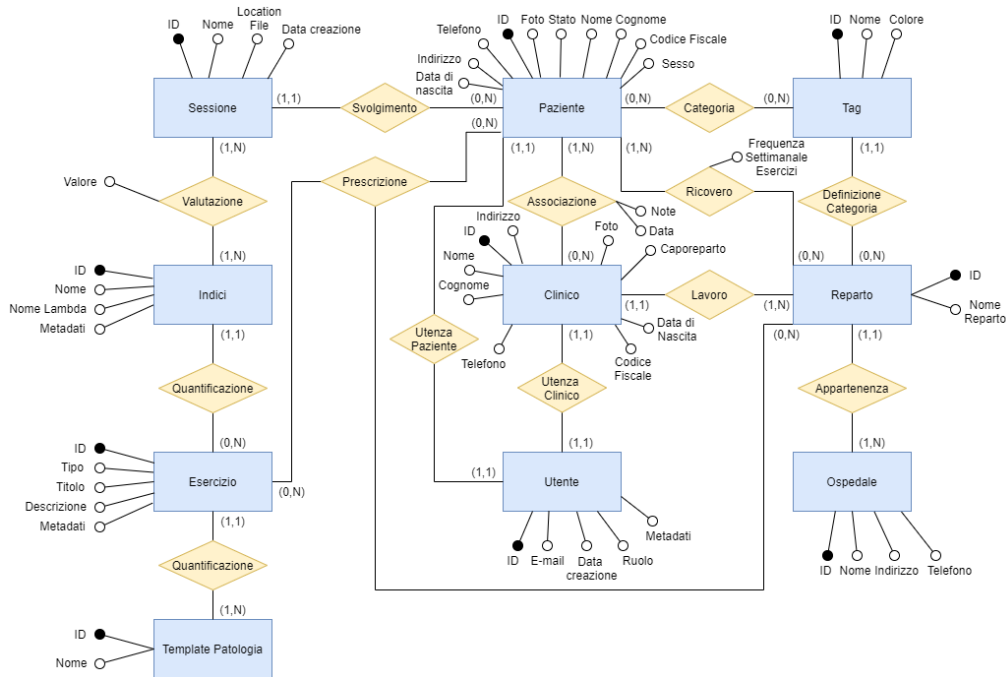


Figura 3.9: Schema E-R del database di sistema

Analizzando lo schema E-R è possibile notare le entità che le costituiscono, che sono:

- **Paziente:** rappresenta i dati di un paziente della piattaforma, i cui attributi sono dati principalmente dalle informazioni anagrafiche, oltre che dalla foto (rappresentata come un URL relativa alla foto del paziente, caricata all'interno del *bucket* dei pazienti) e allo stato in cui si trova, usato dall'applicazione del clinico per catalogare i pazienti. Nel contesto del sistema ogni paziente è associato ad un'utenza, può essere associato a uno o più pazienti e reparti, e può essere catalogato tramite l'uso di zero o più *tag*, utili per operazioni di filtraggio. Il paziente, inoltre, può essere associato a uno o più esercizi da svolgere per ciascun reparto.
- **Clinico:** rappresenta i dati di un clinico della piattaforma. Gli attributi che lo compongono sono principalmente dati dalle informazioni anagrafiche, e da un attributo che indica se il clinico è il caporeparto del reparto a cui

afferisce. Ogni clinico è associato ad un'utenza dell'applicazione, segue zero o più pazienti ed è assegnato ad un unico reparto.

- **Reparto:** rappresenta i dati di un reparto clinico o ospedaliero, in cui la piattaforma è abilitata. Ogni reparto è associato ad un unico ospedale e definisce uno o più tag, usate per catalogare i pazienti.
- **Ospedale:** rappresenta i dati di una struttura ospedaliera o clinica, a cui sono associati uno o più reparti che la compongono, e che sono abilitati per l'utilizzo della piattaforma.
- **Tag:** rappresenta un *tag*, definito a livello di reparto e utilizzato per catalogare i pazienti. Ogni *tag* può essere associato a un colore, che ne permetterà una semplice visualizzazione a livello di applicazione.
- **Sessione:** rappresenta i dati relativi ad una sessione svolta da un paziente. Ad ogni sessione possono essere associate delle note, la data di esecuzione e un URL contenente il percorso, nel *bucket* S3, in cui è memorizzato il file che contiene i dati multimediali associati alla sessione, consultabili dal clinico. Ogni sessione è associata ad un unico paziente e contiene diversi indici, associati agli esercizi svolti al suo interno.
- **Template Patologia:** è un'entità che rappresenta una patologia. Collegando una patologia a un insieme di esercizi, viene offerta quindi al clinico la possibilità di scegliere dei *preset* che contengono al loro interno degli esercizi adatti per una patologia, offrendo allo stesso tempo la possibilità di personalizzare gli esercizi da effettuare per il singolo paziente.
- **Esercizio:** rappresenta i dati relativi ad un esercizio che deve essere svolto da un paziente. Ad ogni esercizio è associato un titolo, una descrizione che esplicita come l'esercizio deve essere svolto, e un campo che indica la tipologia di dato associato all'esercizio (audio o video). Ogni esercizio può quindi essere associato a più indici, che rappresentano la performance associata all'esercizio.

- **Indice:** rappresenta i dati relativi ad un indice di un esercizio, che sarà calcolato tramite un algoritmo di analisi audio o video. Per quanto riguarda l'indice, vengono memorizzati principalmente il nome e la funzione lambda a cui è associato il calcolo (usato dall'orchestratore per determinare a quale consumatore ridirezionare il file multimediale). Ogni indice è associato ad un esercizio ed a diverse sessioni. In particolare, nella relazione che lega l'indice alla sessione, viene memorizzato il risultato della valutazione dell'indice.

Lo schema E-R è corredato di una tavola dei volumi, presente nella Tabella 3.1, che rappresenta le cardinalità associate alle relazioni molti a molti e alle entità dello schema concettuale.

Le operazioni principali che agiscono sul database sono le classiche operazioni di creazione, ricerca, modifica ed eliminazione di uno o più elementi appartenente a ciascuna entità/relazione, oltre alle operazioni di selezione di paziente in base al reparto e alle *tag* specificate, di selezione degli indici relativi ad un paziente ordinati temporalmente, necessario per costruire la *dashboard* per visualizzare l'andamento della patologia del paziente, e di selezione delle sessioni svolte da un paziente.

3.4.2 Scelta del modello logico e del database

Dopo aver definito lo schema concettuale si è quindi passati alla fase di scelta del database da adottare per il progetto. In questa fase, sono stati analizzati i volumi di dati per verificare la presenza di eventuali colli di bottiglia che potessero giustificare la scelta di un database NoSQL piuttosto che un database relazionale, per garantire una maggior disponibilità e scalabilità.

Seppur la maggior parte delle entità e relazioni non presentino cardinalità elevate, essendo l'applicazione usata da un'utenza ristretta, è stato individuato un possibile problema nella scalabilità della relazione che lega l'entità Indice a Sessione per la memorizzazione dei risultati degli indici di una sessione, che presenta una cardinalità molto elevata, come è possibile vedere nella Tabella

Tavola dei volumi	
Entità / Relazione	Cardinalità
Paziente	500
Clinico	20
Utente	520
Tag	60
Reparto	5
Ospedale	3
Template Patologia	1
Esercizio	30
Sessione	30000
Indice	30
Paziente - Clinico	1500
Paziente - Reparto	700
Sessione - Indice	900000
Indice - Esercizio	30
Paziente - Tag	2000

Tabella 3.1: Tavola dei volumi relativa alle entità e alle relazioni presenti nello schema E-R

3.1. Per verificare se tale relazione potesse essere problematica nell'utilizzo di un database relazionale è stato effettuato uno stress test all'interno di un database PostgreSQL, in cui sono state definite le tabelle associate ai pazienti, alle sessioni, agli indici, agli esercizi e alle relazioni che legano queste entità, popolandole con una grande quantità di dati per verificare l'eventuale degrado delle performance delle operazioni all'aumentare dei dati.

In particolare, sono stati individuati due scenari. Nel primo scenario, che presenta un carico di lavoro più vicino a quello reale dell'applicazione, sono supposti:

- 500 pazienti, che svolgono sessioni nel corso di un anno intero;

CAPITOLO 3. PAROLA DI MOTONEURONE: PROGETTAZIONE DELL'ARCHITETTURA CLOUD E DEL DATABASE

- 30 esercizi, con 1 indice per esercizio;
- 5 sessioni per mese;
- un periodo di operatività pari a un anno.

In questo scenario, la relazione che potrebbe presentare un collo di bottiglia ha una cardinalità pari 900000 tuple.

Nel secondo scenario, che presenta un carico di lavoro ampiamente più alto di quello supposto per l'applicazione, sono supposti:

- 5500 pazienti, che svolgono sessioni nel corso di un anno intero;
- 30 esercizi, con 1 indice per esercizio;
- 5 sessioni per mese;
- un periodo di operatività pari a un anno.

In questo caso, la cardinalità della relazione incriminata pari a 9900000 tuple.

Sono stati quindi misurati, per entrambi i scenari, i tempi di risposta delle seguenti operazioni:

1. Selezione di tutti i valori degli indici;
2. Selezione di tutti i dati risultanti dal join tra le tabelle associate ai valori degli indici, alle sessioni, ai pazienti, agli indici e agli esercizi;
3. Selezione di tutti i dati descritti dall'operazione 2, filtrati per id del paziente;
4. Selezione di tutti i dati descritti dall'operazione 2, filtrati per id del paziente e per id dell'indice;
5. Selezione di tutti i dati descritti dall'operazione 2, filtrati per id della sessione.

	Scenario 1	Scenario 2
Operazione 1	300 ~ 450 ms	10 ~ 15 s
Operazione 2	2 s	18 s
Operazione 3	100 ~ 200 ms	450 ~ 500 ms
Operazione 4	50 ~ 100 ms	50 ~ 100 ms
Operazione 5	100 ms	100 ms

Tabella 3.2: Tabella che presenta i risultati dello stress test, raffigurante i tempi di risposta medi associati alle operazioni per entrambi gli scenari descritti

I risultati dell'analisi, effettuata su un portatile di fascia media, con un processore Ryzen 2500U, 8GB di RAM, e 256GB di SSD, sono presenti nella Tabella 3.2. Come è possibile vedere, le prime due operazioni presentano un notevole incremento del tempo di risposta a causa della necessità di effettuare lo *scan* su una grande quantità di dati. La terza operazione, invece, risulta essere solo in parte impattata, con un tempo di risposta raddoppiato, mentre la quarta e la quinta operazione non presentano cambiamenti sostanziali. Considerato il fatto che le prime tre operazioni non sono di interesse per l'applicazione, mentre le ultime due operazioni sono quelle effettivamente utilizzate nel sistema, è possibile concludere che per le esigenze della piattaforma l'uso di un database relazionale non presenta particolari problemi di scalabilità, per cui non è necessaria l'adozione di un database NoSQL.

Il *database management system* scelto, in particolare, è stato PostgreSQL, avendo constatato la sua maturità e le funzionalità offerte. In particolare, è stata ritenuta molto utile la possibilità, da parte del DBMS, di integrare dei campi in formato `jsonb` all'interno di una relazione. Questo ha consentito di definire dei campi aggiuntivi su entità quali Esercizio, Indice o Utente, in cui vengono specificati attributi opzionali e aggiuntivi integrati in seguito all'aggiunta di nuove funzionalità in un contesto agile, garantendo a queste relazioni una maggior flessibilità. Dopo aver constatato l'utilità dei nuovi attributi sarà poi possibile effettuare una riprogettazione del database, in cui verranno inclusi nel nuovo database progettato.

Nel contesto dell'architettura cloud realizzata su AWS, sarà usato il servizio Amazon Relational Database Service (RDS) per implementare il database PostgreSQL.

3.4.3 Progettazione logica e fisica

Per terminare la progettazione del database del sistema si è quindi proceduti con la progettazione logica e fisica in cui, a partire dallo schema E-R, sono state ottenute le relazioni finali da implementare nel database, corredati di indici primari e secondari.

Le relazioni prodotte in seguito a questa fase sono illustrate nella Tabella 3.3. Per quanto riguarda gli indici associati al database, infine, si è deciso di introdurre degli indici primari su tutte le chiavi primarie di ciascuna tabella, mentre è stato definito un indice secondario sulla data di una sessione, consentendo di effettuare selezioni e ordinamenti per data efficientemente allo scalare del dataset.

Relazione
OSPEDALE (<u>id</u> : int, nome: string, indirizzo: string, telefono: string)
REPARTO (<u>id</u> : int, nome: string, ospedale: int)
TAG (<u>id</u> : int, nome: string, colore: string, reparto: int)
UTENTE (<u>id</u> : int, email: string, data_creazione: timestamp, ruolo: string, metadata: jsonb)
PAZIENTE (<u>id</u> : int, account: int, nome: string, cognome: string, codice_fiscale: string, telefono: string, indirizzo: string, sesso: char, stato: string, foto: string, data_nascita: date)
CATEGORIA (<u>paziente</u> : int, <u>tag</u> : int, data_creazione: timestamp)
OSPEDALIZZAZIONE (<u>paziente</u> : int, <u>reparto</u> : int, frequenza_esercizi: int)
CLINICO (<u>id</u> : int, account: int, reparto: int, nome: string, cognome: string, codice_fiscale: string, telefono: string, indirizzo: string, foto: string, data_nascita: date)
FOLLOW (<u>paziente</u> : int, <u>clinico</u> : int, nota: string, data_nota: timestamp)
SESSIONE (<u>id</u> : int, data_creazione: timestamp, paziente: int, location_file: string)
RISULTATO_INDICE (<u>sessione</u> : int, <u>indice</u> : int, risultato: jsonb)
INDICE (<u>id</u> : int, esercizio: int, nome: string, nome_lambda: string, metadata: jsonb)
ESERCIZIO (<u>id</u> : int, patologia: int, titolo: string, tipo: string, descrizione: string, metadata: jsonb)
PATOLOGIA (<u>id</u> : int, nome: string)
PRESCRIZIONE (<u>paziente</u> : int, <u>reparto</u> : int, <u>esercizio</u> : int)

Tabella 3.3: Relazioni da implementare nel database, ottenuti in seguito alla fase di progettazione logica

Capitolo 4

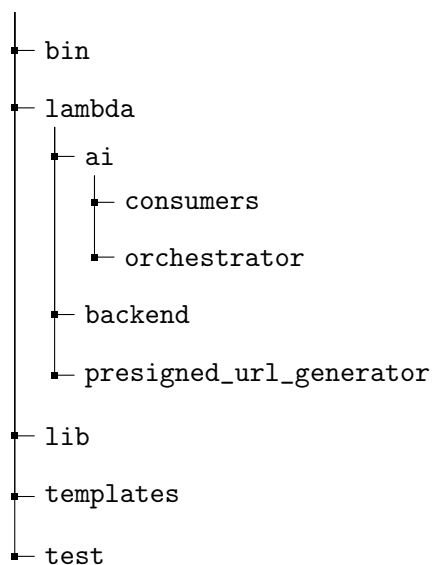
Sviluppo del progetto

In questo capitolo verrà trattata la fase di sviluppo del progetto, che segue la fase di progettazione trattata nel capitolo precedente. Verranno descritti, in particolare, gli strumenti utilizzati per lo sviluppo di ciascuna delle componenti fondamentali del progetto e le scelte di implementazione effettuate, al fine di rendere il sistema flessibile e facilmente estendibile.

4.1 Sviluppo infrastruttura

Per quanto riguarda lo sviluppo e il *deploy* dell'architettura cloud progettata è stato usato il *framework* AWS Cloud Development Kit (CDK), al fine di definire le risorse associate al progetto in maniera programmatica. In questo modo, è possibile garantire una facile manutenibilità ed estendibilità dell'architettura: ogni modifica all'infrastruttura genererà un *change set* che descriverà le modifiche da implementare in maniera incrementale, senza la necessità di modificare le risorse manualmente.

Il progetto per la definizione dell'infrastruttura è stato sviluppato in TypeScript e NodeJS, ed è suddiviso nelle seguenti cartelle:



In particolare le cartelle più rilevanti del progetto sono le seguenti:

- **bin**: contiene il punto di ingresso dell'applicazione eseguita da CDK, in cui sono definiti gli aspetti di rilievo per l'esecuzione del *framework* quali ad esempio l'*environment*, utile per specificare la regione e l'account da usare per effettuare la distribuzione dell'architettura su AWS.
- **lib**: è la componente focale del progetto che contiene le definizioni dei costrutti che compongono l'architettura, usate da CDK per generare uno *stack* in CloudFormation che può essere usato per il *deploy*.
- **templates**: contiene dei modelli di base che possono essere usati per lo sviluppo di una funzione lambda. Un esempio di *template* è quello relativo al consumatore, contenente del codice da integrare all'algoritmo che consente il corretto funzionamento della funzione lambda. Questo *template*, in particolare, verrà descritto in una delle prossime sezioni.
- **lambda**: contiene al suo interno il codice associato a ciascuna delle funzioni lambda che devono essere integrate nell'architettura. Come è possibile vedere, questa cartella è suddivisa in diverse sottocartelle che contengono il codice associato alle singole componenti: in particolare, nella cartella **backend** e **presigned_url_generator** sono presenti i codici relativi al *backend*

del sistema e alla funzione per il caricamento dei file su S3, mentre la cartella `orchestrator` contiene il codice associato all'orchestratore, e infine `consumers` contiene a sua volta una serie di sottocartelle, una per ciascuno dei consumatori da implementare su una funzione lambda.

- `test`: cartella che contiene il codice necessario per effettuare delle operazioni di *unit testing* sull'architettura realizzata.

All'interno della cartella `lib`, quindi, sono state definite le componenti necessarie per il corretto funzionamento dell'infrastruttura. Le componenti sono dichiarate all'interno di una classe che, estendendo la classe `cdk.Stack`, consente la definizione di uno *stack* di CloudFormation. Le componenti che sono state definite, quindi, sono:

- La funzione lambda associata all'orchestratore, che si occupa di gestire la distribuzione dei file multimediali per l'analisi.
- Le funzioni lambda associate ai consumatori, che si occupano di elaborare gli indici tramite algoritmi di intelligenza artificiale.
- Le code SQS associate alle funzioni lambda, che consentono la distribuzione dei messaggi relativi ai file da elaborare.
- La funzione lambda che costituisce il *backend* del servizio, che sarà sviluppato in un contesto *serverless* per offrire una buona flessibilità e scalabilità.
- I *bucket* necessari per il funzionamento del sistema, che sono quello relativo ai pazienti e quello usato come *storage* temporaneo dei file multimediali per la computazione degli indici.
- La funzione lambda usata per la generazione degli *URL presigned*, necessaria per il caricamento del file di una sessione nel *bucket* dei pazienti, tramite la generazione di un link con una data scadenza per l'*upload*.
- I servizi di API Gateway per l'interfacciamento dei *client* con le lambda per il *backend* e per la generazione dell'*URL presigned*.

- Gli *event source mapping*, che consentono l'esecuzione automatica delle funzioni lambda in seguito all'avvenimento di un evento. In particolare è stato definito un *event source mapping* sull'orchestratore, che si mette in ascolto sul *bucket* dei pazienti per individuare la creazione di nuovi file associati alle sessioni, e su ciascuno dei consumatori, che si mettono in ascolto sulla coda associata, avviando la computazione nel caso in cui vengono ricevuti dei messaggi.

Oltre alle componenti sopra descritte sono anche stati definiti i permessi associati alle funzioni lambda, per permettere il loro interfacciamento con le altre risorse presenti all'interno del sistema. In particolare:

- Alla funzione lambda associata all'orchestratore sono stati garantiti i permessi di lettura e scrittura sul *bucket* del paziente, ed il permesso di inviare messaggi su ciascuna delle code relative ai consumatori.
- Alle funzioni lambda associate ai consumatori sono stati garantiti i permessi di lettura e di eliminazione di un file sul *bucket* usato per lo *storage* temporaneo dei file da elaborare, e i permessi di consumo dei messaggi della coda associata al particolare consumatore.
- Alla funzione usata per la generazione degli *URL presigned* è stato garantito il permesso di scrittura all'interno del *bucket* del paziente, per permettere la generazione del link per il caricamento del file.
- A tutte le funzioni lambda in generale è stato aggiunto il ruolo `AWSLambdaVPCAccessExecutionRole`, che consentirà l'associazione di una Virtual Private Network (VPC) ad una funzione lambda.

Il database, al contrario delle altre risorse qui descritte, non è stato implementato direttamente all'interno dello *stack*, ma è stato definito esternamente tramite la console di AWS. Il motivo di questa scelta è insita nella maggior facilità di gestione del database in questo modo, garantendo una migliore governance e flessibilità nella gestione dei relativi account. In particolare, una volta

definito il database sono stati iniettati all'interno delle funzioni lambda il nome dell'account del database e la password tramite l'uso di variabili di ambiente, per garantire l'interfacciamento diretto. Oltre al database, inoltre, è stata definita anche una Virtual Private Cloud apposita. Tale rete è stata configurata in modo tale da rendere il database accessibile esclusivamente alle funzioni lambda che ne fanno uso, evitando accessi dall'esterno per ridurre la superficie di attacco e aumentare la sicurezza del sistema. Per garantire la possibilità delle funzioni lambda di interfacciarsi ad Internet, pur partecipando alla rete privata, è stato necessario inoltre definire una NAT Gateway associato alla VPC, per consentire la comunicazione del sistema verso l'esterno.

Una funzionalità chiave realizzata all'interno del progetto è data dal sistema di caricamento dinamico dei consumatori. Questo sistema consente, all'avvio del progetto, di individuare tutte le funzioni definite per i consumatori, inserite all'interno di una cartella apposita, per permettere la definizione in maniera dinamica dei consumatori con le relative code senza la necessità di definirle esplicitamente nel codice. In questo modo viene garantita una facile estendibilità al progetto così definito, in quanto per definire un nuovo consumatore è sufficiente definire una nuova cartella in cui inserire il codice da eseguire, e il progetto genererà automaticamente la funzione lambda e la coda associata.

Per realizzare questa funzionalità è stata definita una classe, visibile in Figura 4.1, che modella le risorse necessarie per definire un consumatore. Tale classe definisce, per ogni istanza, la funzione lambda relativa ad un consumatore, con la coda SQS associata e tutti i permessi necessari per il corretto funzionamento. Per l'istanziamento di un consumatore vengono passate al costruttore delle proprietà, composte da:

- Le impostazioni relative alla funzione lambda da creare;
- Un dato booleano per indicare se il codice associato alla lambda dovrà essere inserito in un container Docker o meno. In questo modo è possibile separare le lambda che necessitano dell'uso di un container da quelle che

```
1 export interface LambdaAIConsumerProps {
2     lambdaSettings: lambda.FunctionProps | lambda.
      DockerImageFunctionProps,
3     unzipBucket: Bucket,
4     useContainer: boolean
5 }
6
7 export class LambdaAIConsumer extends cdk.Construct {
8     consumer!: lambda.Function;
9     queue!: sqs.Queue;
10
11     constructor(scope: cdk.Construct, id: string, props:
      LambdaAIConsumerProps) {
12         super(scope, id);
13
14         this.consumer = props.useContainer ?
15             new lambda.DockerImageFunction(this, id, props.
      lambdaSettings as lambda.
      DockerImageFunctionProps) :
16             new lambda.Function(this, id, props.lambdaSettings
      as lambda.FunctionProps);
17
18         this.queue = new sqs.Queue(this, id + "-queue", {
19             queueName: id + "-queue"
20         });
21
22         this.consumer.addEventSource(new SqsEventSource(this.queue, {
23             enabled: false
24         }));
25
26         this.consumer.role?.addManagedPolicy(ManagedPolicy.
      fromAwsManagedPolicyName("service-role/
      AWSLambdaVPCAccessExecutionRole"));
27
28         this.queue.grantConsumeMessages(this.consumer.role as IRole);
29
30         props.unzipBucket.grantDelete(this.consumer.role as IRole);
31         props.unzipBucket.grantRead(this.consumer.role as IRole);
32     }
33 }
```

Figura 4.1: Codice per la definizione di un consumatore. Se istanziato, verrà definito un nuovo consumatore, composto da una lambda function e dalla relativa coda

invece non ne fanno uso, definendo dinamicamente la tipologia di istanza da allocare con le relative impostazioni;

- Il riferimento al *bucket* per lo *storage* temporaneo dei dati da elaborare.

Nella Figura 4.2, invece, è illustrato il codice che implementa il sistema di caricamento dinamico dei consumatori. Questo sistema elenca tutte le cartelle presenti nel percorso `lambda/ai/consumers`, in cui sono presenti i codici da caricare su ciascuna lambda, e controlla l'eventuale presenza del Dockerfile per determinare se il consumatore da definire fa uso di un container Docker o meno. In base a questa informazione è possibile quindi definire delle impostazioni specifiche per la lambda, che verranno usate in fase di istanziamento del consumatore. Per ogni cartella viene quindi creata un'istanza del consumatore, che verrà inclusa nello *stack* da caricare su AWS.

4.2 Sviluppo delle funzioni lambda

Dopo aver parlato dello sviluppo dell'infrastruttura su AWS, è quindi possibile parlare dello sviluppo delle singole componenti che compongono il sistema. In questa sezione, in particolare, si parlerà dello sviluppo delle funzioni lambda usate nell'infrastruttura, quali i consumatori, l'orchestratore e il generatore di *URL presigned* per il caricamento di file sul *bucket* S3 relativo ai pazienti. Il *backend*, essendo una componente focale del progetto, verrà trattato a parte nella prossima sezione.

Bisogna notare, a tal riguardo, che le funzioni lambda qui presentate sono state implementate nel linguaggio Python. In particolare, sono state usate librerie quali `boto3` [20], per l'interfacciamento con gli altri servizi, quali S3 e SQS, e `psycopg2` [21], per l'interfacciamento con un database PostgreSQL.

4.2.1 Lambda per il caricamento degli esercizi su bucket

In questa sezione si illustrerà, per iniziare, una prima implementazione della funzione lambda usata per il caricamento degli esercizi sul *bucket* dei pazienti,

```
1  const consumerDirs = fs.readdirSync("./lambda/ai/consumers");
2  for (let consumerDir of consumerDirs) {
3    let useContainer = fs.existsSync('./lambda/ai/consumers/${
4      consumerDir}/Dockerfile');
5    let assetOptions = useContainer ?
6      {
7        code: lambda.DockerImageCode.fromImageAsset('./lambda/ai/
8          consumers/${consumerDir}', {
9            cmd: ["lambda_function.lambda_handler"]
10           });
11         code: lambda.Code.fromAsset('./lambda/ai/consumers/${
12           consumerDir}'),
13         runtime: lambda.Runtime.PYTHON_3_8,
14         handler: "lambda_function.lambda_handler",
15       };
16     let consumerLambda = new LambdaAIClientConsumer(this, `ai-consumer-${
17       consumerDir}`, {
18       unzipBucket: unzippedBucket,
19       useContainer: useContainer,
20       lambdaSettings: {
21         ...assetOptions,
22         memorySize: 4096,
23         timeout: cdk.Duration.seconds(900),
24         vpc: vpc,
25         vpcSubnets: vpc.selectSubnets({
26           subnetType: ec2.SubnetType.PRIVATE
27         }),
28         securityGroups: [securityGroup],
29         environment: {
30           ...databaseLambdaEnv,
31           UNZIP_BUCKET_NAME: unzippedBucket.bucketName,
32           ACCOUNT_ID: props?.env?.account as string,
33           REGION: props?.env?.region as string
34         }
35       }
36     });
37     consumerLambda.queue.grantSendMessages(orchestrator.role as IRole);
38   }
39 }
```

Figura 4.2: Codice per la definizione dinamica dei consumatori, situati all'interno di una cartella.

tramite la generazione di un *URL presigned*.

Il codice che implementa questa funzione è presente nella Figura 4.3. Come è possibile vedere la lambda riceve una richiesta di caricamento da parte di un *client*, che specifica il *filename* e l'identificatore del paziente associato. Conseguentemente viene definito il percorso associato al file da caricare nel *bucket*, il quale verrà inserito all'interno di una cartella che ha come nome l'identificatore associato al paziente stesso. In seguito, vengono effettuati dei controlli per verificare che il file non esista già all'interno del *bucket*, per evitarne la sovrascrittura, e viene generato l'*URL presigned*, con una scadenza pari a 10 minuti, che sarà inviato come risposta al *client*. Il *client* potrà quindi effettuare l'*upload* del file tramite una richiesta HTTP di tipo PUT per il suo inserimento sul *bucket* dei pazienti.

4.2.2 Lambda orchestratore

Un'altra funzione lambda rilevante è quella che svolge il ruolo dell'orchestratore, che si occupa di effettuare la distribuzione dei file multimediali tra diversi consumatori per l'invocazione degli algoritmi di analisi.

Uno snippet del codice dell'orchestratore, che indica la parte focale dell'esecuzione, è disponibile nella Figura 4.4. La funzione rimane in ascolto per il caricamento di nuovi file all'interno del *bucket* dei pazienti. Nel momento in cui viene caricato un nuovo file compresso, contenente i dati della sessione, si attiva la funzione, che preleva il file all'interno del *bucket* per poi aprirlo tramite la libreria *zipfile*. Se l'apertura del file zip avviene con successo viene invocata la funzione *init_session_and_get_indexes*, che si occupa di scrivere sul database la nuova sessione, con *timestamp* pari a quello di ricezione del file, e di restituire le associazioni tra gli esercizi e gli indici, sotto forma di dizionario, e l'insieme di tutti gli indici con le informazioni associate. A questo punto è possibile iniziare con la distribuzione: per ognuno dei file inclusi nello zip viene individuato l'esercizio di riferimento che è presente all'interno del nome del file, espresso nel formato `<id esercizio>_filename`. Dopo aver trovato l'esercizio è quindi possibile procedere con l'ottenimento degli indici associati, collegati a un consumatore, usando il

```
1 import boto3
2 import os
3 import json
4
5 PATIENT_BUCKET = os.environ['PATIENT_BUCKET']
6
7 def key_existing_size__list(client, key):
8     """return the key's size if it exist, else None"""
9     response = client.list_objects_v2(
10         Bucket=PATIENT_BUCKET,
11         Prefix=key,
12     )
13     for obj in response.get('Contents', []):
14         if obj['Key'] == key:
15             return obj['Size']
16
17 def lambda_handler(event, context):
18     # Get the service client.
19     s3 = boto3.client('s3')
20     body = json.loads(event["body"])
21     filename = body["filename"]
22     patient_id = body["patient_id"]
23
24     path = str(patient_id) + "/" + filename
25
26     if key_existing_size__list(s3, path):
27         return {
28             "error": "File already exists!"
29         }
30
31     presigned_url = s3.generate_presigned_url(
32         ClientMethod='put_object',
33         Params={
34             'Bucket': PATIENT_BUCKET,
35             'Key': path
36         },
37         ExpiresIn=600
38     )
39
40     return {
41         "statusCode": 200,
42         "body": json.dumps({
43             "upload_url": presigned_url
44         })
45     }
```

Figura 4.3: Snippet del codice che implementa il generatore per l'url presigned

```

1 def init_session_and_get_indexes(patient_id, file_location):
2     ...
3
4 def get_queue_url(queue_name):
5     return "https://sqs." + REGION + ".amazonaws.com/" + ACCOUNT_ID + "/"
6         " + queue_name
7
8 def lambda_handler(event, context):
9     try:
10        bucket_name = event['Records'][0]['s3']['bucket']['name']
11        file_key = event['Records'][0]['s3']['object']['key']
12        [patient_id, _] = file_key.split("/")
13        obj = s3.get_object(Bucket=bucket_name, Key=file_key)
14        zip_file = ZipFile(io.BytesIO(obj['Body'].read()))
15        (session_id, indexes, exercise_to_indexes) =
16            init_session_and_get_indexes(patient_id, file_key)
17    except Exception as e:
18        print("Errore nel processamento dello zip " + file_key + ": "
19            + str(e))
20
21 for filename in zip_file.namelist():
22     try:
23         logger.info(filename)
24         exercise_id, new_filename_extended = filename.split("_")
25         new_filename, extension = new_filename_extended.split(".")
26         index_ids = exercise_to_indexes[int(exercise_id)]
27         for index_id in index_ids:
28             with zip_file.open(filename) as file:
29                 save_filename = new_filename + "_" + str(index_id) + "
30                     ." + extension
31                 logger.info(save_filename)
32                 s3.put_object(Bucket=UNZIP_BUCKET_NAME, Key=
33                     save_filename, Body=file)
34                 sqs.send_message(QueueUrl=get_queue_url(indexes[
35                     index_id]), MessageBody=save_filename,
36                     MessageAttributes={
37                         "session": {
38                             "StringValue": str(session_id),
39                             "DataType": "String"
40                         },
41                         "index": {
42                             "StringValue": str(index_id),
43                             "DataType": "String"
44                         }
45                     })
46     except Exception as e:
47         print("Errore nel processamento del file " + filename + ": "
48             + str(e))

```

Figura 4.4: Snippet del codice che implementa l'orchestratore

dizionario precedentemente descritto che contiene i *mapping* tra gli esercizi e gli indici. Per concludere, quindi, viene effettuato salvataggio del file, per ogni indice, all'interno del *bucket* per lo *storage* temporaneo dei file decompressi, e viene inviato un messaggio alla coda del consumatore associato all'indice. Il messaggio inviato, in particolare, è costituito da un corpo che indica il file da elaborare all'interno del bucket, e da due attributi che indicano, rispettivamente, l'id della sessione e l'id dell'indice di riferimento, usati dal consumatore per il salvataggio del risultato.

4.2.3 Lambda consumatore

Descritto l'orchestratore, che si occupa della distribuzione dei file, è possibile quindi passare alla descrizione dell'implementazione dei consumatori, che si occupano di svolgere effettivamente la computazione degli algoritmi di analisi sui file multimediali.

In questo caso, considerato il fatto che un consumatore ospita un generico algoritmo per la computazione dell'indice, è stato realizzato un *template* che descrive un modello generico di consumatore, integrabile successivamente con l'algoritmo di analisi.

Il codice associato a tale *template* è presente nella Figura 4.5. Come è possibile vedere in questo caso vi è solo lo scheletro della funzione, che dovrà essere integrato con l'algoritmo di analisi. In particolare, è possibile vedere dalla figura come il flusso di funzionamento individuato nel *template* sia il seguente:

1. La funzione lambda viene attivata in seguito all'invio di uno o più messaggi alla coda SQS associata, che saranno estrapolati per l'elaborazione.
2. Per ciascuno dei messaggi ricevuti, vengono estratti il nome del file da elaborare, l'id della sessione di riferimento e l'id dell'indice di riferimento.
3. Avviene il caricamento e il processamento del file dal *bucket*. Questa parte è mancante nel *template*, richiedendo l'integrazione con l'algoritmo di analisi, che dovrà essere invocato all'interno di questa sezione del codice

```
1 import boto3
2 import logging
3 import io
4 import os
5 import psycpg2
6
7 UNZIP_BUCKET_NAME = os.environ['UNZIP_BUCKET_NAME']
8 ACCOUNT_ID = os.environ['ACCOUNT_ID']
9 REGION = os.environ['REGION']
10
11 s3 = boto3.client("s3")
12 logger = logging.getLogger()
13 logger.setLevel(logging.INFO)
14
15 def add_index_result(index_id, session_id, result):
16     conn = psycpg2.connect(
17         host=os.environ['DB_HOST'],
18         database=os.environ['DB_NAME'],
19         user=os.environ['DB_USERNAME'],
20         password=os.environ['DB_PASSWORD'])
21     indexes = {}
22     cur = conn.cursor()
23     cur.execute("INSERT INTO index_result (index_id, session_id, result)
24                 VALUES (%s, %s, %s)", (index_id, session_id, result))
25     logger.info("Added new result " + index_id + " " + session_id)
26     conn.commit()
27     conn.close()
28
29 def lambda_handler(event, context):
30     records = event["Records"]
31     for record in records:
32         try:
33             filename = record['body']
34             session_id = record['messageAttributes']['session']['stringValue']
35             index_id = record['messageAttributes']['index']['stringValue']
36
37             # PROCESSAMENTO DATO
38             result = {
39                 'value': 1
40             }
41             add_index_result(index_id, session_id, result)
42             s3.delete_object(Bucket=UNZIP_BUCKET_NAME, Key=filename)
43         except Exception as e:
44             # HANDLE ERROR
45             logger.info(e)
46             s3.delete_object(Bucket=UNZIP_BUCKET_NAME, Key=filename)
```

Figura 4.5: Template di base da cui è possibile partire per lo sviluppo di un consumatore, integrando l'algoritmo di analisi al suo interno.

per effettuare l'elaborazione sul file, con il conseguente ottenimento del risultato.

4. Viene determinato il risultato finale, che sarà inserito all'interno del database tramite la funzione `add_index_result`, che si occupa di inserire il risultato all'interno del database, nella tabella associata.
5. Viene eliminato il file dal *bucket*, essendo terminata la sua elaborazione.

Nel caso in cui avviene un errore durante l'elaborazione del dato, viene effettuato un log dell'errore riscontrato e si procede comunque all'eliminazione del file, per evitare di avere file non necessari all'interno del *bucket*.

4.3 Sviluppo del backend

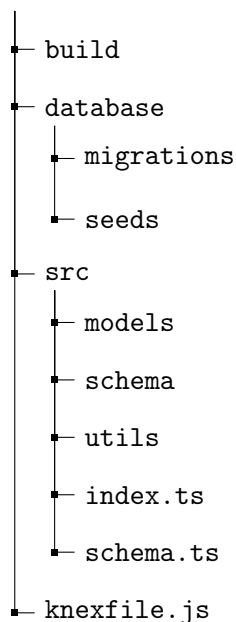
Per terminare la discussione sull'implementazione del sistema si parlerà ora dello sviluppo del *backend* e del database associato.

Il *backend* del progetto è implementato in un contesto *serverless*, facendo uso di una funzione lambda. In questo modo vengono ridotti i costi di operazione dell'architettura (in quanto il *backend* non viene chiamato molto frequentemente, comportando dei bassi costi di invocazione della lambda) e viene garantita una scalabilità automatica del servizio in base alle richieste effettuate, grazie all'approccio *serverless*. Per quanto riguarda l'API esposta dal *backend*, è stato usato il linguaggio di interrogazione **GraphQL**. La scelta di GraphQL deriva principalmente dalla facilità di utilizzo con i *client*, che possono specificare direttamente i dati da ottenere tramite una *query* senza dover contattare multipli *endpoint* per ottenere lo stesso risultato. In questo modo è possibile implementare l'intero *backend* all'interno di una singola funzione lambda, senza dover necessariamente usare diverse funzioni per ogni *endpoint*, come nel caso di una REST API.

Per quanto riguarda l'implementazione del *backend* sono stati utilizzati TypeScript e NodeJS per l'implementazione, essendo questo uno degli ambienti più maturi per la realizzazione di servizi di questa natura. Lo strumento usato per l'implementazione del server, in questo contesto, è **Apollo Server**. Que-

sto *framework* consente di definire, in maniera agevole ed efficace, un server che espone un'interfaccia interrogabile tramite *query* che seguono lo standard di GraphQL. Questo *framework* è applicabile anche in un contesto *serverless*, tramite l'uso della libreria `apollo-server-lambda`, per cui l'integrazione del *framework* all'interno di una funzione lambda risulta essere semplice da effettuare. All'interno del progetto del *backend*, inoltre, è stata usata la libreria **Objection.js** per l'Object-Relational Mapping (ORM), che permette di effettuare l'associazione tra i modelli definiti nel *backend* e le tabelle del database relazionale. Il vantaggio nell'uso di tale libreria, rispetto alle altre alternative, sta nella flessibilità consentita: Objection.js [22], infatti, fa uso della libreria **Knex.js** [23], che è un *query builder* per l'interfacciamento con il database. Ciò consente di evitare la rigidità tipica delle tradizionali librerie di ORM, dando la possibilità di realizzare facilmente *query* personalizzate ed adeguatamente sanificate facendo uso della funzionalità di *query building* offerta dalla libreria.

Di seguito è presente la struttura a cartelle adottata per l'organizzazione del progetto del *backend*:



È possibile vedere, ad alto livello, la presenza delle seguenti cartelle:

- **database**: contiene al suo interno gli script usati per inizializzare e popolare il database. Al suo interno è possibile individuare le

cartelle:

- `migrations`, che contiene gli script usati per effettuare le migrazioni del database, consentendo una sua inizializzazione immediata su un nuovo dispositivo.
- `seeds`, che contiene gli script usati per eseguire il *seeding* del database, consentendo di popolarlo con dati *mockup*, utili per lo sviluppo della piattaforma prima del *deploy*.
- `src`: contiene il codice sorgente dell'applicazione. Al suo interno sono presenti i file `index.ts` e `schema.ts` che, si occupano, rispettivamente, di definire il punto di ingresso del *backend* e di inizializzare lo schema relativo al database, integrando tutti gli schemi definiti nella cartella `schema`. Al suo interno sono quindi presenti le seguenti cartelle:
 - `models`, che contiene i modelli usati dall'applicazione, che si interfacciano al database tramite l'utilizzo di una libreria di ORM.
 - `schema`, che contiene il codice sorgente usato per la generazione degli schemi in GraphQL, assieme alle *query* e alle *mutation*.
 - `utils` che contiene le funzioni ausiliarie necessarie per il corretto funzionamento del server di *backend*.
- `build`: contiene i file JavaScript per l'esecuzione del *backend*, ottenuti tramite la compilazione del codice sorgente definito in TypeScript.

È possibile ora parlare dell'implementazione delle componenti più rilevanti del servizio di *backend*, descrivendone nel dettaglio il funzionamento. Verrà descritto innanzitutto lo sviluppo degli script di *migration* e *seeding*, dopodiché saranno illustrati i modelli realizzati e il sistema di generazione automatica degli schemi che è stato definito.

4.3.1 Migrazione e seeding del database

Il primo passo per la realizzazione del *backend* è consistito nella creazione di diversi script di *migration* e *seeding*, che permettono, rispettivamente, di istanziare il

```
1 exports.up = function(knex) {
2   return knex.schema
3     .createTable("patient", function(table) {
4       table.increments("id").notNull().primary();
5       table.integer("user_account_id").notNull().references("id").
6         inTable("user_account").onUpdate("CASCADE").onDelete("
7           CASCADE");
8       table.string("name", 30).notNull();
9       table.string("surname", 30).notNull();
10      table.date("birth_date").notNull();
11      table.specificType("fiscal_code", "CHAR(16)").notNull().
12        unique();
13      table.string("telephone", 15).notNull();
14      table.string("address", 40).notNull();
15      table.specificType("sex", "CHAR").notNull();
16      table.string("status", 20).notNull();
17      table.string("photo", 500);
18    });
19 };
20
21 exports.down = function(knex) {
22   return knex.schema.dropTable("patient");
23 };
```

Figura 4.6: Esempio di script per effettuare una migrazione, che consente la creazione della tabella associata ai pazienti

database tramite modifiche incrementali e di popolarlo con dei dati di *mockup*. In particolare è stata usata la libreria Knex.js per effettuare entrambe le operazioni, tramite i comandi `knex migrate` e `knex seed`, avendo definito il file `knexfile.js` contenente le configurazioni del database su cui eseguire le operazioni. La definizione delle *migration* e delle operazioni di *seeding* avviene tramite la definizione di diversi script, che identificano dei passi che verranno eseguiti sequenzialmente per la realizzazione delle due procedure. In particolare, i script relativi alle *migration* sono posizionati nella cartella `database/migrations`, mentre quelli per il *seeding* sono posizionati nella cartella `database/seeds`.

Degli esempi di script di *migration* e *seeding* sono visibili nelle Figure 4.6 e 4.7.

Per quanto riguarda le *migration*, come è possibile vedere dallo script, vengono esposte due funzioni `up` e `down` che consentono, rispettivamente, di eseguire la modifica e di disfarla, consentendo il *rollback* dell'operazione. Al suo interno,

```
1 const faker = require("faker")
2
3 exports.seed = async function(knex) {
4   await knex('patient').del()
5   let patients = [];
6   let userIds = await knex.select("id")
7     .from("user_account")
8     .where({role: "patient"});
9
10  for(let el of userIds) {
11    patients.push({
12      user_account_id: el.id,
13      name: faker.name.firstName(),
14      surname: faker.name.lastName(),
15      birth_date: faker.date.between('1920-01-01', '2004-12-31'),
16      fiscal_code: faker.random.alphaNumeric(16),
17      telephone: faker.phone.phoneNumber().slice(0, 15),
18      address: (faker.address.streetAddress() + ", " + faker.address.city
19        ()).slice(0, 40),
20      sex: faker.random.arrayElement(['M', 'F']),
21      status: "ricoverato",
22      photo: faker.image.avatar()
23    });
24  }
25  await knex("patient").insert(patients);
26  };
```

Figura 4.7: Esempio di script per effettuare il seeding della tabella del database associata ai pazienti

quindi, è possibile usare la libreria `knex` per effettuare la creazione di nuove relazioni oppure la modifica di tabelle già esistenti.

Per quanto riguarda il *seeding* viene esposta invece la funzione `seed`, in cui viene popolata una relazione. In questo caso è stata usata anche la libreria `faker.js` per la generazione di dati di *mockup*, utile per generare facilmente grandi quantità di dati su cui è possibile effettuare sperimentazioni per il *backend*.

4.3.2 Sviluppo dei modelli del backend

Per consentire una corretta implementazione del servizio di *backend* è stato innanzitutto necessario definire i modelli dei dati richiesti dal sistema, garantendo un corretto interfacciamento al database. Per la definizione dei modelli, come già descritto in precedenza, è stato usato l'ORM di `Object.js`. Tramite l'uso di questa libreria, infatti, è possibile definire il modello associato ad una tabella del database tramite la creazione di una classe apposita contenente le proprietà di rilievo, con l'eventuale definizione di vincoli di integrità, rappresentati tramite un JSON Schema, e di vincoli di chiave esterna tra un modello e un altro. Questi vincoli di chiave esterna, definiti tramite la proprietà `relationMappings`, definiscono esplicitamente le relazioni tra diversi modelli con la relativa tipologia (molti a molti, uno a molti, uno a uno). In questo modo è possibile realizzare delle *query* che effettuano operazioni di *join* tra diverse relazioni, i cui risultati vengono innestati all'interno di un'unica struttura a grafo¹. Questa struttura si presta particolarmente bene per l'interfacciamento dei modelli con gli schemi definiti in GraphQL.

Per quanto riguarda la creazione dei modelli, quindi, è stata innanzitutto definita una classe `BaseModel`. Questa classe, usata come base per la costruzione dei modelli, contiene le proprietà e i metodi rilevanti che devono essere ereditati da questi. La classe è visibile nella Figura 4.9, di cui sono rappresentati i dettagli essenziali. Come è possibile vedere, sono presenti proprietà quali `externalTypeMapping` ed `externalSchemasDefinition`, necessarie, come si vedrà in seguito, per la generazione automatica degli schemi per ciascun modello. Nel caso del

¹<https://vincit.github.io/object.js/api/query-builder/eager-methods.html>


```
1 export interface SchemaDefinition {
2   name: string;
3   jsonSchema: JSONSchema;
4   externalTypesMapping: Object;
5   isInput?: boolean;
6 }
7
8 export interface ResolverMetadata {
9   name: string,
10  schema: string;
11  resolver: IFieldResolver<any, any, any>;
12 }
13
14 export class BaseModel extends Model {
15   static columnNameMappers = snakeCaseMappers();
16   static externalTypesMapping = {};
17   static externalSchemasDefinition: SchemaDefinition[] = [];
18
19   ...
20
21   static getGraphQLQuery(info: GraphQLResolveInfo) {
22     let fields = graphqlFields(info);
23     let joinFields = this.getJoinFields(fields);
24     let result = this.query().withGraphJoined(joinFields);
25     return result;
26   }
27
28   static getJoinFields(fields) {
29     let graphQuery = {};
30     Object.keys(this.relationMappings).forEach(e1 => {
31       if (fields[e1]) {
32         if (fields[e1] == {}) {
33           graphQuery[e1] = true;
34         } else {
35           graphQuery[e1] = this.relationMappings[e1].modelClass.
36             getJoinFields(fields[e1]);
37         }
38       }
39     });
40     return graphQuery;
41   }
42 }
```

Figura 4.8: Snippet del codice per la definizione della classe BaseModel, ereditata da ciascun modello

`BaseModel` questi elementi sono vuoti, ma se necessario i modelli che derivano da questa classe ridefiniranno queste proprietà. È inoltre presente il metodo statico `getGraphQLQuery` che, date le informazioni su una *query* in GraphQL inviata da un *client*, si occupa di estrarne i campi richiesti e di effettuare il *join* tra le relazioni coinvolte dalla richiesta dinamicamente, al fine di ottenere il risultato finale da restituire al *client*.

Nella Figura 4.8 invece è presente, a titolo di esempio, il modello di un utente dell'applicazione, la cui classe è derivata da `BaseModel`. La classe contiene:

- Le definizioni delle proprietà associate al modello;
- La proprietà statica `tableName` che definisce la tabella relativa al modello, consentendo l'interfacciamento con il database;
- Il JSON Schema che definisce le proprietà da rispettare per i campi del modello, usato da `Object.js` per effettuare la validazione del dato in fase di inserimento o aggiornamento nel database;
- Le proprietà statiche `externalTypesMapping` e `externalSchemasDefinition`, il cui ruolo verrà esplicitato nella fase di generazione degli schemi;
- La proprietà statica `relationalMappings`, che definisce i vincoli di chiave esterna tra il modello corrente ed altri modelli. In questo esempio è possibile vedere, in particolare, la definizione della relazione uno ad uno tra l'utente e il paziente associato che definirà una proprietà `patient` all'interno del modello, che può essere popolata tramite i metodi `withGraphQLJoined` o `withGraphQLFetched`, restituendo la struttura a grafo precedentemente menzionata.

4.3.3 Generazione automatica degli schemi dai modelli

Avendo definito i modelli richiesti per il funzionamento del *backend* è ora necessario definire gli schemi, le *query* e le *mutation* associate. Nel contesto di GraphQL, in particolare:

```
1 export interface UserMetadata {
2     status: string;
3 }
4
5 export class User extends BaseModel {
6     id: number;
7     email: string;
8     role: string;
9     creationDate: string;
10    metadata?: UserMetadata;
11    patient?: Patient;
12    clinician?: Clinician;
13
14    static tableName = "user_account";
15    static jsonSchema = {
16        type: "object",
17        properties: {
18            id: {type: "integer"},email: {type: "string", minLength: 0,
19                maxLength: 30},
20            role: {type: "string", minLength: 0, maxLength: 10},
21            creationDate: {type: "string", format: "date-time"},
22            metadata: {type: "object"},
23        },
24        required: ["email", "role", "creationDate"]
25    };
26    static externalTypesMapping = {
27        metadata: "UserMetadata"
28    };
29    static externalSchemasDefinition = [{
30        name: "UserMetadata",
31        jsonSchema: {properties: {status: {type: "string"}}},
32        externalTypesMapping: {},
33        isInput: true
34    }];
35    static get relationMappings() {
36        return {
37            patient: {
38                relation: Model.HasOneRelation,
39                modelClass: Patient,
40                join: {
41                    from: "user_account.id",
42                    to: "patient.user_account_id"
43                }
44            },
45            ...
46        };
47    }
48 }
```

Figura 4.9: Codice per la definizione del modello dell'utente

- Lo schema definisce i tipi di dati che possono essere restituiti da una *query* o *mutation* di GraphQL, tramite una notazione che fa uso di una tipizzazione stretta. Questi schemi sono necessari sia allo sviluppatore, che in questo modo comprende come utilizzare le *query* offerte dal servizio, che al servizio di *backend* stesso, che userà lo schema come strumento di validazione per il dato, sia in lettura che in scrittura.
- Le *query*, definite all'interno dello schema `Query`, definiscono le richieste che possono essere inoltrate al server per ottenere i dati di rilievo. Per ciascuna *query* vengono quindi specificati i parametri, per eseguire filtri sui risultati, e il tipo di ritorno, che stabilisce cosa viene restituito dal server.
- Le *mutation*, definite all'interno dello schema `Mutation`, permettono invece di eseguire operazioni di scrittura all'interno del database, come ad esempio l'inserimento di una risorsa, il suo aggiornamento o la sua cancellazione. Similmente alle *query*, anche le *mutation* hanno la presenza di parametri e di un tipo di ritorno.

Un semplice esempio di schema in GraphQL, comprensivo di *query* e *mutation*, è visibile nella Figura 4.10.

Considerati i requisiti di flessibilità ed estendibilità del sistema, in cui si richiede la possibilità di modificare velocemente i modelli del *backend* e le relative relazioni sul database per aggiungere delle funzionalità alla piattaforma, si è deciso di implementare un sistema di generazione automatica di schemi, *query* e *mutation* associati ai modelli definiti nel *backend*. Questi schemi possono essere eventualmente estesi dallo sviluppatore tramite la definizione di *query* e *mutation* personalizzate. In questo modo ogni modifica effettuata ai modelli comporta una modifica automatica degli schemi associati, senza la necessità di modifiche manuali.

In particolare per ciascuno dei modelli vengono generati, all'avvio del *backend*:

- gli schemi che definiscono il tipo di dato associato al modello in GraphQL;

```
1 type Book {
2   title: String
3   author: Author
4 }
5
6 type Author {
7   name: String
8   books: [Book]
9 }
10
11 type Query {
12   books: [Book]
13   authors: [Author]
14 }
15
16 type Mutation {
17   addBook(title: String, author: String): Book
18 }
```

Figura 4.10: Snippet che rappresenta uno schema in GraphQL, corredato di query e mutation

- le *query* per selezionare un insieme di istanze o un'istanza specifica del modello;
- le *mutation* che permettono di aggiungere, aggiornare o eliminare un'istanza specifica del modello.

Per la generazione degli schemi è stata usata una classe apposita, chiamata `SchemaGenerator`. Questa classe contiene dei metodi statici che consentono, per un modello, la generazione in maniera automatica degli elementi appena descritti.

In Figura 4.11 è presente uno snippet di un metodo statico di questa classe che, presi in input la classe associata al modello e un oggetto contenente le *query* e *mutation* aggiuntive definite dal programmatore, restituisce lo schema, contenente la definizione del tipo del modello e delle *query* e *mutation* associate, e i *resolver*, che specificano le operazioni che il *backend* deve effettuare in corrispondenza di una determinata *query* o *mutation*. In particolare, all'interno di questa funzione vengono invocati il metodo `generateSchemasFromModel` per ottenere gli schemi relativi al modello, e i metodi `generateQueriesAndResolvers` e `gene-`

```
1 static generateSchemaAndResolvers(model: typeof BaseModel, resolversObj:
  {Query?: ResolverMetadata[], Mutation?: ResolverMetadata[]} = {}):
  {schema: DocumentNode, resolvers: IResolvers} {
2   let typeDef = this.generateSchemasFromModel(model);
3   let {querySchema, queryResolvers} = this.generateQueriesAndResolvers
    (model, resolversObj.Query);
4   let {mutationSchema, mutationResolvers} = this.
    generateMutationsAndResolvers(model, resolversObj.Mutation);
5   let schema = gql`
6     ${typeDef}
7     ${querySchema}
8     ${mutationSchema}
9     `
10  let resolvers = {...queryResolvers, ...mutationResolvers};
11  return {schema: schema, resolvers: resolvers}
12 }
```

Figura 4.11: Snippet della funzione statica `generateSchemaAndResolvers`, che consente la generazione degli schemi, query e mutation associati a un modello

`rateMutationsAndResolvers` per generare sia gli schemi che i *resolver* associati alle *query* e alle *mutation* definite per il modello.

Per quanto riguarda la funzionalità di generazione degli schemi associati al modello, uno scheletro del metodo `generateSchemasFromModel` è visibile in Figura 4.12. Come è possibile vedere dalla figura, nella funzione vengono generati una serie di schemi tramite l'uso ripetuto del metodo `generateSchemaFromSingleModel`, applicato sia al modello stesso che ad eventuali schemi esterni, definiti sempre all'interno del modello. È stato necessario definire questi schemi esterni aggiuntivi, tramite la proprietà `externalSchemasDefinition` del modello, per garantire la capacità di definire degli schemi anche per i campi in formato `jsonb` di una relazione, che si presentano in un formato semistrutturato. È infatti necessario definire degli schemi in GraphQL anche per questi tipi di dato, in quanto non è possibile definire un tipo che descriva un generico oggetto semistrutturato. Il metodo `generateSchemaFromSingleModel`, non riportato nella sua totalità per motivi di brevità, si occupa invece di generare una stringa che definisce lo schema GraphQL associato a un modello o ad uno schema esterno. La generazione dello schema avviene facendo uso di diversi elementi contenuti all'interno della classe o interfaccia, quali:

```
1 private static generateSchemasFromModel(model: typeof BaseModel): string
  {
2   let baseModelSchema = this.generateSchemaFromSingleModel(model);
3   let additionalSchemas = model.externalSchemasDefinition.map(el =>
      this.generateSchemaFromSingleModel(el));
4   return [baseModelSchema, ...additionalSchemas].join("\n");
5 }
6
7 private static generateSchemaFromSingleModel(model: typeof BaseModel |
  SchemaDefinition): string {
8   ...
9 }
```

Figura 4.12: Snippet della funzione statica `generateSchemaAndResolvers`, che consente la generazione degli schemi, query e mutation associati a un modello

- Il JSON Schema associato al modello o schema esterno, che definisce le proprietà che lo compongono, con i tipi associati. In questo modo, è possibile effettuare una semplice conversione tra i tipi di dato presenti nel JSON Schema e i tipi primitivi di GraphQL, per la definizione delle proprietà associate allo schema.
- Il campo `externalTypesMapping`, che definisce esplicitamente il tipo per alcune delle proprietà composte (ad esempio, le proprietà associate ad un campo espresso nel formato `jsonb` della relazione). A queste proprietà, infatti, non può essere assegnato un tipo di dato che indichi un oggetto generico, ma è necessario indicare tipi di dato già definiti nello schema GraphQL (eventualmente, i tipi associati gli schemi esterni che sono stati definiti appositamente per i campi `jsonb`).
- Il campo `relationalMappings` di un modello, usato per definire all'interno dello schema i campi che definiscono le relazioni tra modelli, che verranno eventualmente popolati in base alla *query* effettuata.

Nella Figura 4.13, invece, è presente uno snippet che indica il funzionamento del metodo `generateQueriesAndResolvers`. In questa funzione si determinano, in primo luogo, i parametri da associare alla *query* per la ricerca delle istanze tramite il valore di uno o più attributi, dopodiché vengono determinate le chiavi

```
1 private static generateQueriesAndResolvers(model: typeof BaseModel,
  additionalResolvers: ResolverMetadata[] = []): {querySchema: string,
  queryResolvers: IResolvers} {
2     let className = model.name;
3
4     let params = ...
5
6     let keys = typeof model.idColumn === "string" ? [model.idColumn]
      : [...model.idColumn];
7
8     let queryModel = `
9     extend type Query {
10         ${className.toLowerCase()}s(${params.join(", ")}): [${
11             className}]
12         ${className.toLowerCase()}(${keys.map(el => `${el}: ID!`).
13             join(", ")}): ${className}
14         ${additionalResolvers.map(el => el.schema).join("\n")}
15     }
16 `
17
18     let resolvers = {
19     Query: {
20         ['${className.toLowerCase()}s']: async (parent, args,
21             context, info) => {
22             ...
23             return res;
24         },
25         ['${className.toLowerCase()}']: async (parent, args,
26             context, info) => {
27             ...
28             return res;
29         },
30         ...additionalResolvers.reduce((acc, el) => {return {...acc
31             , [el.name]: el.resolver}}, {})
32     }
33 };
34
35     return {querySchema: queryModel, queryResolvers: resolvers}
36 }
```

Figura 4.13: Snippet della funzione statica generateSchemaAndResolvers, che consente la generazione degli schemi, query e mutation associati a un modello

primarie della relazione, al fine di contrassegnarle correttamente come id nella definizione della *query*. Vengono definiti, quindi, gli schemi associati alle *query* per la selezione di elementi multipli e specifici, e ad eventuali altre *query* che sono state definite esternamente dal programmatore. Sono definiti, infine, i *resolver* associati a ciascuna *query*: a questo fine sono state scritte delle funzioni che consentono di effettuare una selezione su multiple istanze oppure su una singola istanza della relazione. A queste funzioni sono inclusi, inoltre, i *resolver* associati alle altre *query* esterne specificate dallo sviluppatore. Viene quindi restituito un oggetto contenente sia lo schema che i *resolver* associati alle *query*.

Il metodo `generateMutationsAndResolvers`, che si occupa di generare gli schemi e i *resolver* per le *mutation*, funziona allo stesso modo.

Una volta definito il meccanismo di generazione degli schemi, è quindi possibile procedere con il loro utilizzo su ciascuno dei modelli del *backend*. Nella cartella `schema` del progetto sono presenti tutti i codici sorgenti in cui questa funzionalità viene invocata per ciascuno dei modelli, con l'eventuale definizione di *query* e *mutation* aggiuntive. Nella Figura 4.14 è visibile un esempio, in cui viene usata la classe `SchemaGenerator` per generare gli schemi e i *resolver* associati al modello dell'indice, con l'aggiunta di una *query* personalizzata che consente di effettuare una selezione per paziente e per indice. Gli schemi e i *resolver* vengono quindi esportati, e in particolare nel file `schema.ts`, presente nella cartella radice del progetto, viene effettuata l'integrazione degli schemi e dei *resolver* esportati di ciascun modello, ottenendo lo schema finale che è infine esportato ed utilizzato nel server, all'interno del file `index.ts`.

```
1 import { Index } from "../models/Index";
2 import { SchemaGenerator } from "../utils/schema_generator";
3
4 const Query = SchemaGenerator.generateSchemaAndResolvers(Index, {
5   Query: [
6     {
7       name: "indexesByPatientAndIndexId",
8       schema: "indexesByPatientAndIndexId(id: ID!, patient: ID!):
9         Index",
10      resolver: async (parent, args, context, info) => {
11        let res = await Index
12          .query()
13          .withGraphJoined('sessions(selectById, orderByDate).
14            patient')
15          .modifiers({
16            selectById(builder) {
17              builder.where({ patient_id: args.patient })
18            },
19            orderByDate(builder) {
20              builder.orderBy("creation_date")
21            }
22          })
23          .findById(args.id);
24        return res;
25      }
26    ]
27  });
28
29 export const typeDefs = Query.schema;
30
31 export const resolvers = Query.resolvers;
```

Figura 4.14: Esempio di script per effettuare una migrazione, che consente la creazione della tabella associata ai pazienti

Capitolo 5

Risultati

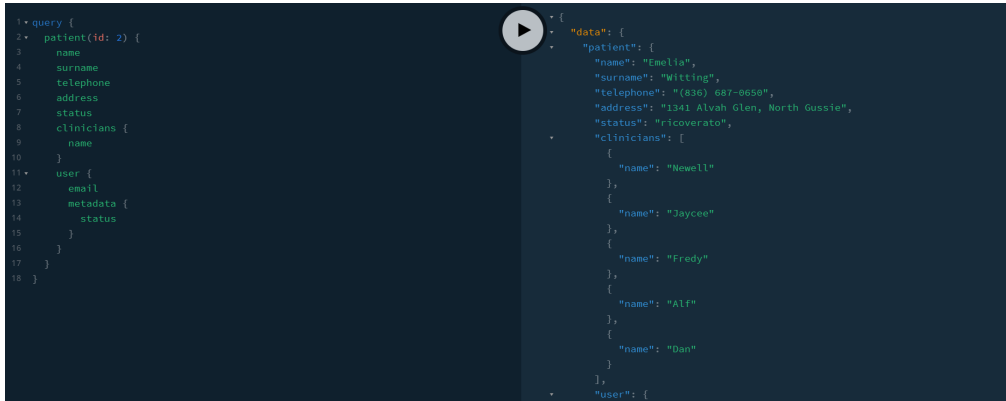
In questa sezione si parlerà dei risultati ottenuti in seguito alla fase di sviluppo, avendo effettuato il *deploy* dell'infrastruttura e dei servizi sviluppati all'interno di un ambiente AWS.

Per quanto riguarda il *deploy* dell'architettura è stata innanzitutto creata un'istanza PostgreSQL, con una dimensione iniziale dello *storage* pari a 20 GB, con dimensionamento automatico e con *backup* automatici settimanali, tramite il servizio Amazon RDS. Il database è stato successivamente inizializzato e popolato tramite gli script di *migration* e *seeding* definiti per il *backend*. Dopo aver fatto ciò, è stato quindi effettuato il *deploy* del resto dell'infrastruttura tramite il *framework* CDK. Una volta definito il progetto e le risorse per la creazione dello *stack*, la distribuzione è stata semplice da eseguire: tramite il comando `cdk deploy`, infatti, è stato generato lo *stack* da caricare all'interno di CloudFormation, e dopo aver dato conferma dell'operazione il *deploy* è stato effettuato in maniera automatica da CDK, senza la necessità di intervenire manualmente durante l'operazione.

Il *backend*, implementato all'interno di una funzione lambda, è stato testato in seguito al *deploy* con esito positivo, dopo essersi accertati di abilitare il meccanismo di *Cross-Origin Resource Sharing* (CORS) per l'interfacciamento con i *client*. Nello specifico, il *backend* implementato offre un *endpoint* che offre la possibilità di definire, testare ed eseguire delle *query* o *mutation* in GraphQL.

CAPITOLO 5. RISULTATI

Questa funzionalità, il cui utilizzo è visibile nella Figura 5.1, risulta essere molto utile per lo sviluppo delle applicazioni che si interfacciano al *backend*, offrendo la possibilità di testare le *query* esplicitamente. Chiaramente in fase di produzione questa interfaccia verrà disattivata, per evitare inutili esposizioni del *backend* verso l'esterno.



```
* query {
1 patient(id: 2) {
2   name
3   surname
4   telephone
5   address
6   status
7   clinicians {
8     name
9   }
10  user {
11   email
12   metadata {
13     status
14   }
15 }
16 }
17 }
18 }
```

```
{
  "data": {
    "patient": {
      "name": "Emilia",
      "surname": "Witting",
      "telephone": "(836) 487-8660",
      "address": "1341 Alvah Glen, North Gussie",
      "status": "ricoverato",
      "clinicians": [
        {
          "name": "Newell"
        },
        {
          "name": "Jaycee"
        },
        {
          "name": "Fredy"
        },
        {
          "name": "Alf"
        },
        {
          "name": "Dan"
        }
      ],
      "user": {

```

Figura 5.1: Esempio di query in GraphQL all'interno del backend realizzato

Dopo la componente di *backend* del sistema, è stata testata la componente relativa alla distribuzione della computazione sui diversi consumatori. Per effettuare il test sono stati innanzitutto caricati due consumatori semplici, che non effettuano alcuna elaborazione ma restituiscono un risultato fisso, che non dipende dal file considerato, per verificare che il flusso di distribuzione ed esecuzione sia corretto e che i risultati siano caricati correttamente nel database.

```
START RequestId: 123c8453-c5fb-5b05-8148-dd1beec43b5f Version: $LATEST
[INFO] 2021-06-19T17:27:49.871Z 123c8453-c5fb-5b05-8148-dd1beec43b5f Added new result 1 3609
END RequestId: 123c8453-c5fb-5b05-8148-dd1beec43b5f
REPORT RequestId: 123c8453-c5fb-5b05-8148-dd1beec43b5f Duration: 6303.79 ms Billed Duration: 6304 ms Memory Size: 4096 MB Max Memory Used: 417 MB
START RequestId: e3f62f82-a9fd-5569-a9dd-76ca8254c2b2 Version: $LATEST
[INFO] 2021-06-19T17:29:26.222Z e3f62f82-a9fd-5569-a9dd-76ca8254c2b2 Added new result 1 3610
END RequestId: e3f62f82-a9fd-5569-a9dd-76ca8254c2b2
REPORT RequestId: e3f62f82-a9fd-5569-a9dd-76ca8254c2b2 Duration: 6217.29 ms Billed Duration: 6218 ms Memory Size: 4096 MB Max Memory Used: 417 MB
```

Figura 5.2: Log associato all'esecuzione di un consumatore su diversi file

Dopo aver accertato il corretto funzionamento del sistema, l'architettura è stata quindi testata su un consumatore associato ad uno degli algoritmi che saranno effettivamente utilizzati all'interno del progetto, che calcola un indice indicante il numero delle ripetizioni del fonema "U-I" effettuate da un paziente in 30 secondi,

```
1 import requests
2 import json
3
4 FILENAME = "<nome_file>"
5 LAMBDA_URL = "<endpoint_lambda>"
6
7 def main():
8     r = requests.post(LAMBDA_URL, data=json.dumps({"patient_id": "1", "
9         filename": FILENAME}))
10    upload_url = r.json()["upload_url"]
11    print(upload_url)
12    with open(FILENAME, "rb") as f:
13        s = requests.put(upload_url, data=f.read())
14        print(s)
15        print(s.content)
16
17 if __name__ == "__main__":
18     main()
```

Figura 5.3: Script di test utilizzato per verificare il corretto funzionamento del sistema di upload distribuito su AWS

a partire da un file audio. L'algoritmo, facendo uso di diverse librerie di grandi dimensioni tra cui pytorch, è stato inserito in un container Docker per garantirne l'esecuzione in un contesto *serverless*. Il container Docker, in particolare, è stato inserito all'interno del servizio Amazon Elastic Container Registry (ECR) [24], offerto da Amazon per il caricamento dell'immagine del container. L'esito del *deploy* del consumatore, dopo una definizione corretta del container, è stato positivo, in quanto il sistema di distribuzione e computazione ha continuato a funzionare senza complicazioni. Per quanto riguarda le prestazioni dell'algoritmo, in Figura 5.2 sono presenti dei log di esecuzione del consumatore su alcuni dei file caricati, ottenuti tramite il servizio CloudWatch di Amazon che permette di monitorare le funzioni lambda e di visionarne i log. Come è possibile vedere il tempo di esecuzione, nel caso in cui la lambda sia già stata avviata, è abbastanza contenuto, pari a circa 6 secondi per ciascun file da elaborare. Nel caso di *cold boot*, invece, i tempi di esecuzione invece si attestano sui 20 secondi per un avvio iniziale, che non comporta sostanziali problemi per la computazione dei file. La quantità di memoria usata, inoltre, non è elevata: come è possibile vedere dalla figura, vengono usati soltanto 427 MB sui 4096 allocati per la funzione lambda.

Per terminare, quindi, è stato testato il sistema di caricamento dei file nel *bucket* S3 dei pazienti. Il test è stato effettuato facendo uso di uno script per simulare il caricamento di un file nel formato zip caricato da parte del *client* del paziente, presente nella Figura 5.3. Lo script effettua una richiesta all'*endpoint* associato alla funzione lambda per la richiesta del caricamento di un file, e alla ricezione della risposta, contenente l'*URL presigned* utilizzabile per l'*upload*, viene fatta una richiesta PUT al link con il file, completando il caricamento. Lo script è stato eseguito con diverse tipologie di file zip, e in tutti casi sono stati ottenuti i risultati attesi, con un caricamento corretto dei dati all'interno del *bucket* S3. Eventuali problemi nell'*upload* del file (a causa, per esempio, di errori di connessioni) saranno gestiti dall'applicazione che verrà implementata lato paziente, che ritenterà il caricamento del file fino a quando l'*upload* non avviene con successo.

Capitolo 6

Conclusione e sviluppi futuri

In questa tesi è stata descritta la progettazione e lo sviluppo dell'architettura cloud e del *backend* relativi al progetto Parola di Motoneurone, che ha l'obiettivo di realizzare una piattaforma di monitoraggio da remoto per pazienti affetti da patologie neurologiche. Il monitoraggio avviene tramite la valutazione della disartria del paziente con algoritmi di intelligenza artificiale e *deep learning*, da cui vengono ottenuti degli indici quantitativi usati per valutare l'evoluzione della malattia del paziente, con la possibilità da parte del clinico di prendere misure adeguate per rallentarne la progressione. In particolare, sono stati illustrati il contesto in cui si inserisce la piattaforma, così come le esigenze che hanno portato alla formulazione del progetto con i relativi obiettivi da soddisfare. Sono state quindi descritte le tecnologie adottate nel corso del progetto, procedendo poi alla descrizione della fase di progettazione, in cui sono stati illustrati i passaggi svolti per la progettazione delle componenti che caratterizzano l'architettura cloud e del database usato per il progetto. È stato successivamente descritto lo sviluppo del progetto, in cui è stata illustrata l'implementazione delle componenti di rilievo dell'architettura cloud e del *backend*, evidenziando l'impatto che hanno avuto le decisioni prese in fase di progettazione per garantire una buona flessibilità ed estensibilità del progetto a nuovi algoritmi di analisi e malattie neurologiche. Per terminare, sono stati mostrati i risultati ottenuti in seguito alla distribuzione del sistema su un ambiente AWS. Il sistema realizzato rispetta tutte le specifiche de-

scritte in fase di progettazione, e costituisce un buon punto d'inizio da cui partire per lo sviluppo e il rilascio del progetto finale.

6.1 Sviluppi futuri

Per quanto riguarda i sviluppi futuri relativi al sistema realizzato, sono state individuate diverse possibili strade di miglioramento da intraprendere.

Un primo sviluppo futuro che è stato individuato riguarda l'implementazione del sistema di autenticazione e la sua integrazione all'interno dell'architettura realizzata. Seppure questo aspetto sia stato trattato in fase di progettazione, infatti, in prima fase la sua implementazione è stata tralasciata, in quanto non rientrava tra gli obiettivi principali da soddisfare inizialmente. Per consentire un corretto interfacciamento con i *client* del paziente e del clinico, tuttavia, sarà necessario anche implementare questo sistema per garantire che i dati della piattaforma siano accessibili solo a chi detiene le appropriate autorizzazioni.

Un altro possibile sviluppo futuro riguarda il sistema di distribuzione dei file per l'elaborazione tramite gli algoritmi di analisi: seppur il sistema realizzato sia flessibile e scalabile in base alle richieste ricevute, bisognerebbe considerare l'eventuale possibilità di adottare algoritmi più complessi di quanto ipotizzato, che possono richiedere, per un batch, tempi di esecuzione anche maggiori del limite previsto per l'esecuzione di una funzione lambda, pari a 15 minuti. Sarebbe possibile, allora, individuare delle piattaforme aggiuntive di computazione da integrare nel sistema che possono superare questa limitazione.

Un altro aspetto da trattare riguarda la possibilità di adottare un sistema di *chunking* dei dati per il caricamento dei file sul *bucket* S3 dei pazienti. In questo modo l'operazione di *upload* di un file viene realizzata tramite il caricamento dei diversi frammenti che lo compongono. Così facendo, in caso di disconnessioni o problemi di *upload*, non è necessario ricominciare il caricamento del file da capo, ma è sufficiente ritentare l'*upload* dei frammenti che non sono stati caricati correttamente.

Altri possibili sviluppi futuri riguardano, infine, il miglioramento del sistema di generazione automatica degli schemi, per permettere la realizzazione di filtri più complessi e per garantire la personalizzazione delle *query* e delle *mutation* definite di default, e il miglioramento della configurazione delle code associate ai consumatori, individuando in particolare delle metodologie per la gestione delle code di tipo *dead letter*, in cui arrivano i messaggi che non sono stati processati con successo dalle funzioni lambda a causa di errori di varia natura nella loro esecuzione.

Elenco delle figure

1.1	Alcuni item della scala di valutazione del profilo Robertson. Si noti come la scala è di natura qualitativa, presentando soltanto quattro possibili valutazioni: Scarso (1), Discreto (2), Buono (3), Ottimo (4). [3]	12
1.2	Alcuni item della scala di autovalutazione della disartria [3] . . .	13
1.3	Sensori utilizzati per l'elettropalatografia, per registrare il comportamento della lingua durante l'eloquio del paziente. L'elettropalatografia combinata alla logopedia convenzionale sono utili nel trattamento delle disartrie moderate e gravi, ma la sensoristica deve essere realizzata ad-hoc per ogni paziente, riducendo la scalabilità e le tempistiche di intervento [6].	15
1.4	Mockup dell'applicazione lato paziente del progetto Parola di Motoneurone	16
1.5	Prototipo della dashboard dell'applicazione web lato clinico del progetto Parola di Motoneurone	17
2.1	Esempio di query nel linguaggio di interrogazione GraphQL, in cui il client specifica nello specifico le informazioni richieste	25
3.1	Use case dell'applicazione del paziente	30
3.2	Use case dell'applicazione del clinico	32
3.3	Sequence diagram che illustra lo svolgimento degli esercizi all'interno dell'applicazione del paziente, con il conseguente caricamento all'interno dello storage	37

3.4	Diagramma che rappresenta l'architettura adottata per la distribuzione e la computazione dei file multimediali tramite algoritmi di analisi basati su tecniche di intelligenza artificiale	41
3.5	Sequence diagram che illustra il procedimento di distribuzione e computazione relativo a una sessione del paziente	42
3.6	Sequence diagram login paziente	43
3.7	Sequence diagram registrazione paziente	44
3.8	Diagramma architetturale dell'applicazione	45
3.9	Schema E-R del database di sistema	46
4.1	Codice per la definizione di un consumatore. Se istanziato, verrà definito un nuovo consumatore, composto da una lambda function e dalla relativa coda	59
4.2	Codice per la definizione dinamica dei consumatori, situati all'interno di una cartella.	61
4.3	Snippet del codice che implementa il generatore per l'url presigned	63
4.4	Snippet del codice che implementa l'orchestratore	64
4.5	Template di base da cui è possibile partire per lo sviluppo di un consumatore, integrando l'algoritmo di analisi al suo interno. . .	66
4.6	Esempio di script per effettuare una migrazione, che consente la creazione della tabella associata ai pazienti	70
4.7	Esempio di script per effettuare il seeding della tabella del database associata ai pazienti	71
4.8	Snippet del codice per la definizione della classe BaseModel, ereditata da ciascun modello	73
4.9	Codice per la definizione del modello dell'utente	75
4.10	Snippet che rappresenta uno schema in GraphQL, corredato di query e mutation	77
4.11	Snippet della funzione statica generateSchemaAndResolvers, che consente la generazione degli schemi, query e mutation associati a un modello	78

4.12	Snippet della funzione statica <code>generateSchemaAndResolvers</code> , che consente la generazione degli schemi, query e mutation associati a un modello	79
4.13	Snippet della funzione statica <code>generateSchemaAndResolvers</code> , che consente la generazione degli schemi, query e mutation associati a un modello	80
4.14	Esempio di script per effettuare una migrazione, che consente la creazione della tabella associata ai pazienti	82
5.1	Esempio di query in GraphQL all'interno del backend realizzato	84
5.2	Log associato all'esecuzione di un consumatore su diversi file	84
5.3	Script di test utilizzato per verificare il corretto funzionamento del sistema di upload distribuito su AWS	85

Elenco delle tabelle

3.1	Tavola dei volumi relativa alle entità e alle relazioni presenti nello schema E-R	49
3.2	Tabella che presenta i risultati dello stress test, raffigurante i tempi di risposta medi associati alle operazioni per entrambi gli scenari descritti	51
3.3	Relazioni da implementare nel database, ottenuti in seguito alla fase di progettazione logica	53

Bibliografia

- [1] Enderby P. Disorders of communication: dysarthria. *Handb Clin Neurol*, 110(273):81, 2013.
- [2] Green Jordan, Yunusova Yana, Kuruvilla Mili, Wang Jun, Pattee Gary, Synhorst Lori, Zinman Lorne, and Berry James. Bulbar and speech motor assessment in als: Challenges and future directions. *Amyotrophic lateral sclerosis & frontotemporal degeneration*, 14(7-8):494–500, 2013.
- [3] Robertson SJ. *Robertson S.J. Disarthria Profile*. Winslow Press, 1982. Versione italiana a cura di Fossi F. e Cantagallo A. Ediz. Omega(1999).
- [4] Barbara Tomik and Roberto J. Guillof Professor. Dysarthria in amyotrophic lateral sclerosis: A review. *Amyotrophic Lateral Sclerosis*, 11(1-2):4–15, 2010.
- [5] Perry Bridget J., Martino Rosemary, Yunusova Yana, Plowman Emily K., and Green Jordan R. Lingual and jaw kinematic abnormalities precede speech and swallowing impairments in als. *Dysphagia*, 33(6):840–847, 2018.
- [6] Kelly Stephen, Main Alison, Manley Graham, and McLean Calum. Electropalatography and the linguagraph system. *Medical engineering & physics*, 22:47–58, 02 2000.
- [7] Amazon Web Services documentation. <https://docs.aws.amazon.com/>.
- [8] AWS Lambda runtimes. <https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html>.

BIBLIOGRAFIA

- [9] Creating Lambda container images. <https://docs.aws.amazon.com/lambda/latest/dg/images-create.html>.
- [10] AWS CDK documentation. <https://docs.aws.amazon.com/cdk/api/v2/>.
- [11] PostgreSQL. <https://www.postgresql.org/>.
- [12] GraphQL documentation. <https://graphql.org/learn/>.
- [13] Apollo Server. <https://www.apollographql.com/docs/apollo-server/>.
- [14] NodeJS. <https://nodejs.org/it/>.
- [15] TypeScript. <https://www.typescriptlang.org/>.
- [16] Python. <https://www.python.org/>.
- [17] Sharing an object with a presigned URL. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html>.
- [18] ResNet | PyTorch. https://pytorch.org/hub/pytorch_vision_resnet/.
- [19] Using AWS Lambda with Amazon SQS. <https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html>.
- [20] Boto3 documentation. <https://boto3.amazonaws.com/v1/documentation/api/latest/index.html>.
- [21] psycopg2. <https://www.psycopg.org/>.
- [22] Objection.js. <https://vincit.github.io/objection.js/>.
- [23] Knex.js. <https://knexjs.org/>.
- [24] Amazon Elastic Container Registry. <https://aws.amazon.com/it/ecr/>.
- [25] Karch Dominik, Kang Keun-Sun, Wochner Katarzyna, Philippi Heike, Hadders-Algra Mijna, Pietz Joachim, and Dickhaus Hartmut. Kinematic assessment of stereotypy in spontaneous movements in infants. *Gait & posture*, 36(2):307–311, 2012.

- [26] Vieira H, Costa N, Sousa T, Reis S, and Coelho L. Voice-based classification of amyotrophic lateral sclerosis: Where are we and where are we going? a systematic review. *Neurodegener Dis*, 19(5-6):163–170, 2019.
- [27] Zayia LC and Tadi P. Neuroanatomy, motor neuron. *StatPearls*, 2020.
- [28] De Marchi F, Sarnelli MF, Solara V, Bersano E, Cantello R, and Mazzini L. Depression and risk of cognitive dysfunctions in amyotrophic lateral sclerosis. *Acta Neurol Scand*, 139:438–445, 2019.
- [29] Logroscino G and Piccininni M. Amyotrophic lateral sclerosis descriptive epidemiology: The origin of geographic difference. *Neuroepidemiology*, (52):93–103, 2019.
- [30] Yunusova Y, Plowman EK, Green JR, Barnett C, and Bede P. Clinical measures of bulbar dysfunction in als. *Frontiers in neurology*, (106), 2019.
- [31] Gabriela M. Stegmann, Shira Hahn, Julie Liss, Jeremy Shefner, Seward Rutkove, Kerisa Shelton, Cayla Jessica Duncan, and Visar Berisha. Early detection and tracking of bulbar changes in als via frequent and remote speech analysis. *npj Digital Medicine*, 3(1):1–5, 2020.
- [32] CDK workshop. <https://cdkworkshop.com/>.