

UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA “ENZO FERRARI”

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

PROVA FINALE

**Cloud computing: Amazon Web Services e distribuzione di
una Django web application**

CANDIDATO:

Davide Gesualdi

MATRICOLA: 119939

RELATORE:

Prof. Nicola Bicocchi

ANNO ACCADEMICO 2019 – 2020

INDICE

Sommario	5
Introduzione.....	6
1. Cloud Computing	9
1.1 Modelli di servizio	12
1.2 Modelli di distribuzione.....	18
1.3 Modello concettuale e ruoli	24
1.4 Il ruolo della virtualizzazione	28
1.5 Architettura a microservizi e container.....	31
1.6 Serverless computing.....	35
2. Strumenti impiegati	41
2.1 Amazon Web Services (AWS)	42
2.1.1 AWS Identity and Access Management (IAM)	46
2.1.2 AWS Elastic Beanstalk.....	47
2.1.3 Amazon Elastic Compute Cloud (EC2).....	48
2.1.4 Amazon Simple Storage Service (S3)	49
2.1.5 Amazon Relational Database Service (RDS)	51
2.1.6 AWS Lambda	52
2.1.7 Amazon Transcribe	53
2.1.8 Amazon Comprehend.....	54
2.2 Django Web Framework.....	56
2.3 Bootstrap Framework	60
3. BrainIO	62
3.1 Interfaccia sistema/utente.....	64
3.2 Struttura progetto Django	72

3.3 Distribuzione su AWS e funzioni Lambda	88
Conclusioni.....	94
Riferimenti.....	97

Sommario

L'elaborato, redatto con l'obiettivo di presentare gli aspetti teorici impiegati nello svolgimento dell'attività progettuale, sotto la supervisione del Professor Nicola Bicocchi, tratta le tematiche legate al paradigma *cloud computing*, soffermandosi sull'implementazione del modello di sviluppo cloud native, *serverless computing*.

Lo sviluppo della web application *BrainIO* richiede l'implementazione dei framework *Django* per la programmazione lato server, *Bootstrap* per il lato client e, la distribuzione su *Amazon Web Services (AWS)*, piattaforma on demand che fornisce servizi di cloud computing.

La prima parte dell'elaborato è incentrata sull'introduzione degli strumenti utilizzati per lo sviluppo dell'applicazione, introducendo appunto i concetti di cloud computing e serverless computing, per poi descrivere i principali servizi d'interesse offerti dalla piattaforma AWS, la struttura del web framework Django e l'impiego del framework Bootstrap per la realizzazione di un responsive web design.

Infine, l'ultima parte dello scritto, riporta una descrizione dettagliata della web application, analizzando le fasi di sviluppo in locale e la distribuzione su cloud, esaminando l'opportuna configurazione delle risorse necessarie per il corretto funzionamento.

Introduzione

Internet ha portato molteplici cambiamenti nel modo in cui vengono utilizzati computer e servizi informatici. Il paradigma maggiormente impiegato è un'architettura client-server, dove il termine client si riferisce al computer generalmente impiegato da singoli individui che richiedono o accedono ad un servizio offerto da un server, contraddistinto dall'elevata capacità di elaborazione, che gli permette un'interazione simultanea con un cospicuo numero di utenti. Storicamente, i mainframe corrispondono a questa descrizione, ma con il progressivo sviluppo dell'industria dei microcomputer e la conseguente diffusione a larga scala, si è arrivati a raggruppare server di elaborazione più piccoli in un rack di server collegati in rete e alloggiati in un edificio chiamato *data center*, congiuntamente ai dispositivi di archiviazione.

L'apice della diffusione dei data center è avvenuto negli anni 1997-2000, con lo scoppio della *bolla delle Dot.com*¹, una bolla speculativa legata alla scoperta delle nuove tecnologie informatiche. Come ogni altra crisi generata da una bolla speculativa, la crisi del Dot.com si è sviluppata attraverso la classica sequenza: estrema fiducia da parte degli investitori nelle potenzialità di un prodotto/azienda, crescita rapida del prezzo del prodotto, evento che fa vacillare le aspettative di importanti guadagni, elevati flussi di vendite, crollo finale del prezzo del prodotto. A partire dal 1994, con la quotazione di Netscape, la società che sviluppò il primo browser commerciale per Internet, si è assistito ad un sorprendente sviluppo di aziende operanti nel settore Internet o, più in generale, nel settore informatico, chiamate *Dot-com companies* (dal suffisso '.com' dei siti attraverso i quali tipicamente tali società operavano). La necessità delle aziende di una connessione ad Internet stabile e veloce e, di una continua operatività, vista l'impraticabilità per molte piccole aziende, ha stimolato la costruzione di edifici molto grandi, chiamati appunto *Internet Data Center*, i quali forniscono funzionalità avanzate, come la possibilità di creare ridondanza nel sistema di comunicazione, caratteristica fondamentale in caso di guasto di uno o più sistemi:

¹ CONSOB. *La bolla delle c.d. Dotcom* [online].

Disponibile su: <<https://www.consob.it/web/investor-education/la-bolla-delle-c.d.-dotcom>> [Data di accesso: 27/01/2021].

John Stewart (2000) *“Se una linea Bell Atlantic viene tagliata, possiamo trasferirli a ... per ridurre al minimo il tempo di interruzione.”*²

Negli ultimi anni, le tecnologie di rete e di archiviazione stanno incrementando sempre più la loro complessità. Di conseguenza, molto arduo è divenuto il compito di gestirle opportunamente da parte dei professionisti in ambito IT (Information Technology). Quindi, un data center pienamente operativo richiede ingenti spese sia a livello economico, sia a livello operativo e, allo stesso tempo, potrebbe non essere completamente utilizzato tutte le ore del giorno, della settimana o dell'anno. Ciò ha indotto a considerare la condivisione delle risorse di un data center con altri utenti di altre organizzazioni, garantendo adeguatamente sicurezza e protezione dei dati e, in particolare, l'isolamento degli utenti. In tal modo è stato possibile distribuire i costi di gestione di un data center su molti utenti.

La maggior parte degli utenti che usufruiscono dei servizi offerti dal server, non si occupa di costruire un data center e non è intenzionata a ridimensionare il proprio business per acquisire questa capacità. Diversamente, è interessata a trasformare la loro applicazione in un prodotto che abbia un valore commerciale. Necessitano, pertanto, di un sistema altamente scalabile, nel caso in cui il loro prodotto diventi improvvisamente popolare, che si astrae dai dettagli sull'implementazione fisica e, in particolare, che isoli l'utente finale da eventuali guasti. Questi sono i fattori chiave alla base della concettualizzazione del paradigma *cloud computing*, reso possibile, principalmente, dalla crescita di Internet, senza il quale sarebbe impensabile l'utilizzo di risorse di elaborazione e archiviazione collocate fisicamente in zone diverse.

Molti utenti di Internet hanno scoperto il fascino del cloud computing direttamente o indirettamente attraverso una varietà di servizi, senza nemmeno conoscere il ruolo che i cloud svolgono nella loro vita. Le vaste risorse computazionali fornite dall'infrastruttura cloud sono impiegate da organizzazioni di ogni tipo, dimensione e settore per una vastissima gamma di casi d'uso, quali backup dei dati, disaster recovery, desktop virtuali, sviluppo e test di software, analisi di Big Data e applicazioni Web. Per esempio, le

² JOHN HOLUSHA, 2000. Commercial Property/Engine Room for the Internet; Combining a Data Center With a 'Telco Hotel'. *The New York Times* [online].
Disponibile su <<https://www.nytimes.com/2000/05/14/realestate/commercial-property-engine-room-for-internet-combining-data-center-with-telco.html>> [Data di accesso: 27/01/2021].

aziende del settore sanitario utilizzano il cloud per sviluppare trattamenti maggiormente personalizzati per i propri pazienti. Le aziende per i servizi finanziari utilizzano il cloud come base dei propri sistemi di rilevamento e prevenzione di attività fraudolente. I realizzatori di videogame utilizzano il cloud per sviluppare videogiochi online per milioni di giocatori in tutto il mondo.

Usufruendo dei servizi offerti da AWS, il fine ultimo dell'attività progettuale, è la realizzazione di una piattaforma software altamente scalabile e modulare, tale da poter essere facilmente ampliata in futuro.

L'applicazione, finalizzata all'impiego in ambito psicologico, è volta a supportare, con un apporto scientifico, le mansioni di prevenzione, consulenza e diagnosi del professionista sanitario, favorendo l'individuazione dello stato emotivo del paziente e facilitando l'analisi dell'andamento terapeutico, fornendo un pratico strumento per una visione grafica, sintetica o dettagliata di una o più sessioni.

1. Cloud Computing

Nel 2011, il *National Institute of Standards and Technology (NIST)* degli Stati Uniti d'America ha definito il *cloud computing* come “un modello che permette, da qualsiasi luogo e in maniera comoda, l'accesso su richiesta tramite rete ad un insieme di risorse di elaborazione condivise e configurabili (es. reti, server, storage, applicazioni e servizi), che vengono rapidamente fornite e rilasciate con il minimo sforzo di gestione o di interazione da parte del fornitore del servizio”. Si riferisce, quindi, ad un nuovo approccio per la fornitura di risorse IT (capacità computazionale, spazio di memorizzazione, applicazioni e altri servizi) sotto forma di servizi accessibili via rete, in modo tale che gli utenti non debbano considerare la posizione fisica del server o dello storage che supporta le loro esigenze.

L'era del cloud computing è iniziata nel 2006, quando Amazon ha rilasciato *Elastic Compute Cloud (EC2)* e *Simple Storage Service (S3)*, i primi servizi forniti dalla piattaforma *Amazon Web Services (AWS)*. Cinque anni dopo, nel 2012, EC2 era già utilizzato in circa 200 paesi e S3 aveva già memorizzato più di 10 miliardi di oggetti. La gamma di servizi offerti dai *Cloud Service Providers (CSPs)* e il numero di utenti è aumentato notevolmente negli ultimi anni.

La definizione del NIST individua cinque caratteristiche principali per il cloud computing:

- *Self-service su richiesta*: l'utente può unilateralmente approvvigionarsi di risorse computazionali, come server, storage o altro, a seconda delle proprie esigenze e in maniera completamente automatica, senza necessità di un'interazione umana con il fornitore di servizi stesso.
- *Ampio accesso in rete*: le risorse sono disponibili in rete e accessibili tramite meccanismi standard che ne promuovono l'uso con piattaforme client diversificate ed eterogenee (es. smartphones, tablets, computer desktop,...) .
- *Condivisione delle risorse*: le risorse di calcolo del provider vengono organizzate per servire più consumatori, utilizzando un modello condiviso (*multi-tenant*), in cui le diverse risorse fisiche e virtuali sono assegnate dinamicamente in base alla domanda. Le risorse offerte sono indipendenti dalla loro locazione fisica, ovvero il cliente non ha controllo o conoscenza dell'esatta locazione fisica delle risorse a lui fornite.

Tuttavia, il provider potrebbe permettere all'utente di specificare dei vincoli sulla locazione delle risorse a lui assegnate in termini di area geografica, Paese o anche singolo data center.

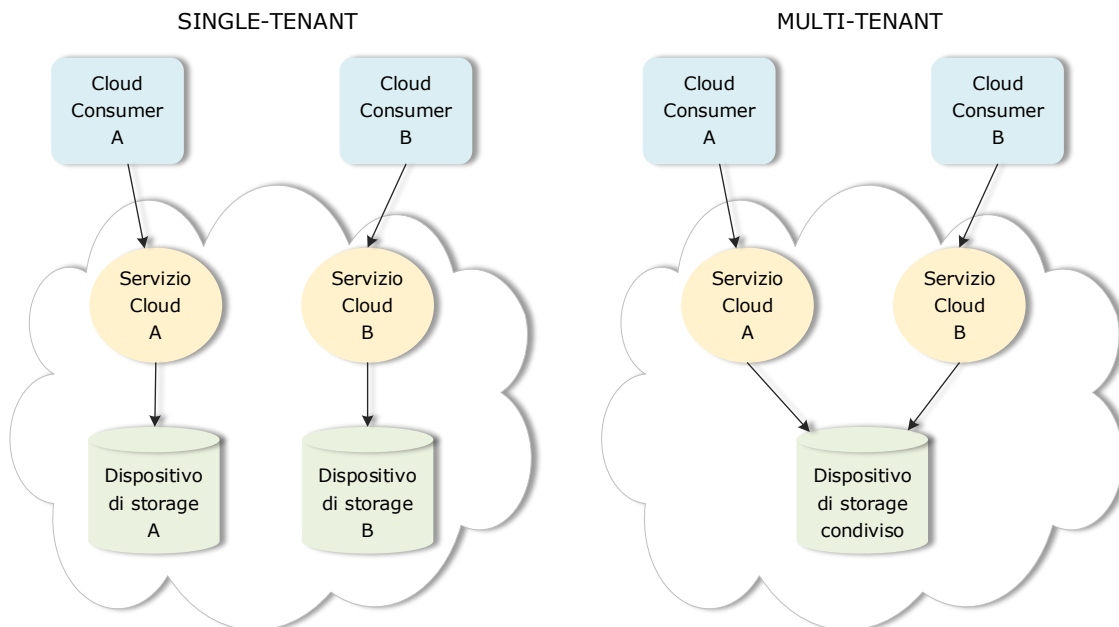


Figura 1.1 – Differenza tra i modelli single-tenant e multi-tenant nella gestione delle risorse cloud.

- *Elasticità*: le risorse possono essere acquisite e rilasciate rapidamente ed elasticamente, in alcuni casi anche automaticamente, per scalare velocemente verso l'alto o verso il basso (aumentare o ridurre la capacità computazionale) in relazione alla domanda. Dal punto di vista dell'utente le risorse appaiono illimitate e, possono essere richieste in qualsiasi quantità e in qualsiasi momento.
- *Servizio misurato*: i sistemi cloud controllano automaticamente e ottimizzano l'uso delle risorse, sfruttando la capacità di misurarne l'utilizzo ad un certo livello di astrazione, appropriato per il tipo di servizio (ad esempio memoria, elaborazione, larghezza di banda, account utente attivi). Il monitoraggio dell'utilizzo dei servizi è molto importante per permettere al provider di reagire ad eventuali picchi di richiesta, allo scopo di garantire al cliente la qualità del servizio promessa. L'utilizzo delle risorse può essere monitorato sia dal provider che dal consumatore, fornendo trasparenza ad entrambi.

Il cloud permette di evitare ingenti spese di capitale (ad esempio, per data center e server fisici) in favore di una spesa variabile, pagando solo le risorse IT realmente consumate. Questa caratteristica garantisce all'utente un notevole risparmio sulle risorse IT, in quanto può ridurre la quantità di risorse elaborative presenti presso le sue strutture e, conseguentemente, il personale per la loro gestione, trasferendo al fornitore di servizi il rischio di inutilizzo delle stesse.

La *resilienza* del cloud computing permette ai Cloud Consumers di incrementare l'affidabilità e la disponibilità delle loro applicazioni. È possibile, infatti, progettare applicazioni che eseguono un failover automatico tra zone di disponibilità diverse senza interruzioni, distribuendo implementazioni ridondanti delle risorse IT tra sedi fisiche diverse.

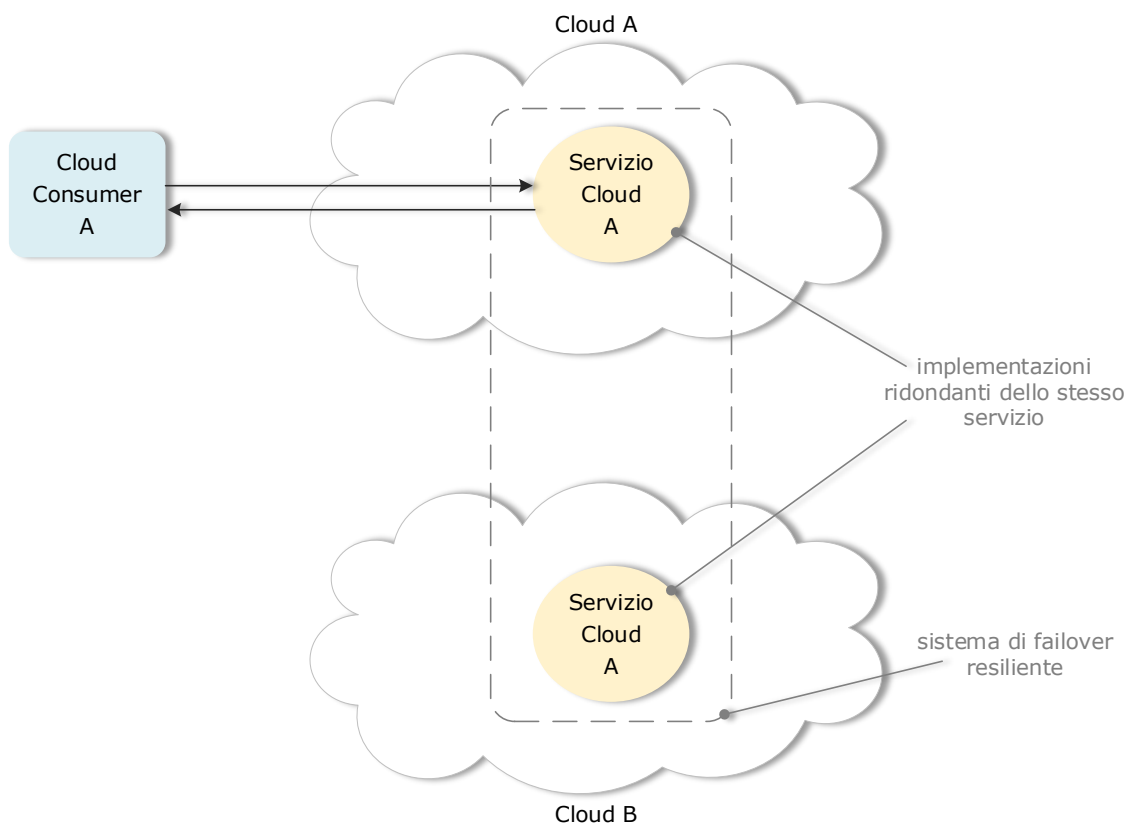


Figura 1.2 – Un sistema resiliente in cui il Cloud B ospita un'implementazione ridondante del Servizio Cloud A, per fornire un failover automatico in caso di indisponibilità del servizio nel Cloud A.

1.1 Modelli di servizio

I servizi di cloud computing possono essere raggruppati in diversi modelli e, a seconda del livello nel quale sono collocati, possono variare i gradi di libertà dell'utente e la complessità della sua interazione con l'infrastruttura cloud. I tre modelli standard nella visione del NIST sono:

- *Infrastructure as a Service (IaaS)*: il Cloud Provider offre la flessibilità di un'infrastruttura fisica, fornendo hardware virtualizzato per l'accesso alle risorse di calcolo, ad esempio spazio virtuale su server, storage e connessione di rete, senza l'onere per l'utente della gestione fisica dell'hardware. Il consumatore non gestisce o controlla l'infrastruttura cloud sottostante, ma ha il controllo sui sistemi operativi, l'archiviazione e le applicazioni distribuite ed eventualmente un controllo limitato sui componenti di rete selezionati (ad es. firewall). Fisicamente, il gruppo di risorse hardware viene estratto da una moltitudine di server e reti, solitamente distribuiti presso numerosi data center, la cui manutenzione è responsabilità del Cloud Provider. Rendendo i servizi di computing disponibili su richiesta, IaaS consente ai consumatori di aumentare o diminuire le risorse a seconda della necessità, pagando solo ciò che usano su base oraria, giornaliera o mensile, riuscendo a gestire adeguatamente le situazioni di picco di attività. Inoltre, può offrire alle aziende l'accesso ad attrezzature e a servizi innovativi e migliorati, come i più recenti processori, hardware di storage e rete, che molte aziende non potrebbero permettersi di acquisire o a cui non sarebbero state in grado di accedere più rapidamente. Tale modello di servizio è tipicamente utilizzato per test e sviluppo, hosting di siti web e app, archiviazione, backup e ripristino, HPC (High Performance Computing). Esempi sono: Amazon Elastic Compute Cloud (EC2), Amazon Simple Storage Service (S3), Amazon Virtual Private Cloud (VPC), Google Compute Engine, Google Cloud Storage.

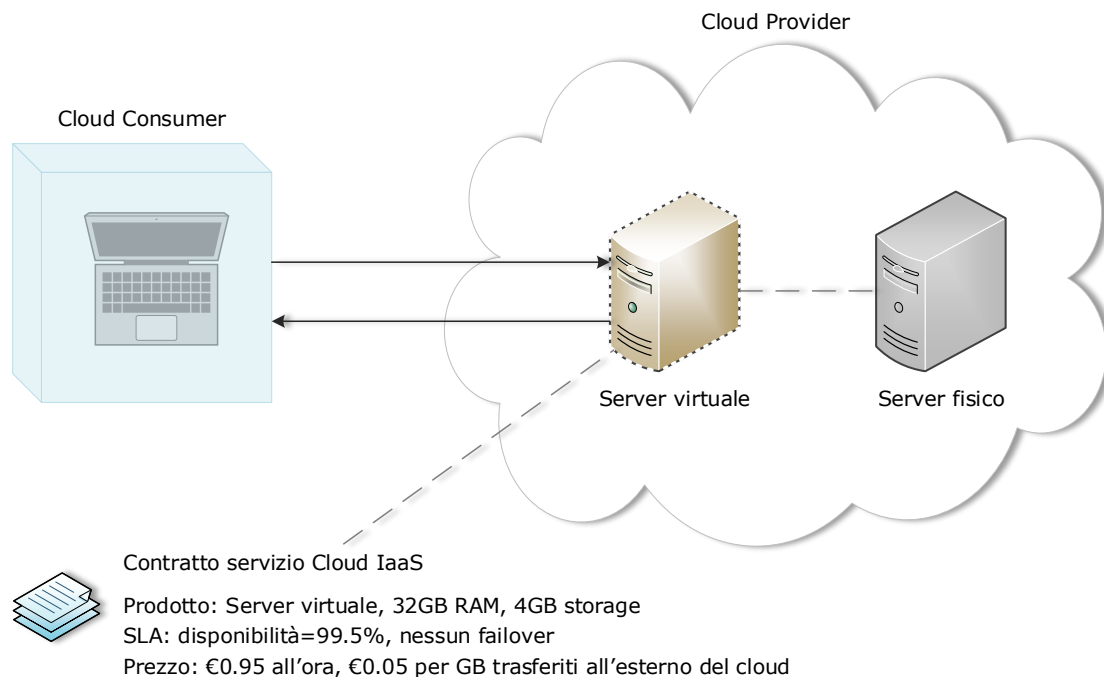


Figura 1.3 – Un Cloud Consumer utilizza un server virtuale all'interno di un ambiente IaaS. I consumatori ricevono una serie di garanzie contrattuali dal Cloud Provider, relative a caratteristiche quali capacità, prestazioni e disponibilità.

- *Platform as a Service (PaaS)*: può essere visto come una piattaforma ponte tra le applicazioni (SaaS) e la parte infrastrutturale (IaaS). PaaS offre la possibilità di distribuire applicazioni create o acquisite da terzi, utilizzando linguaggi di programmazione, librerie, servizi e strumenti supportati dal Cloud Provider. Il consumatore non gestisce né controlla l'infrastruttura cloud sottostante, compresi rete, server, sistemi operativi e memoria, ma ha il controllo sulle applicazioni ed eventualmente sulla configurazione dell'ambiente che le ospita. Va sottolineato che questa tipologia di cloud è dedicata soprattutto agli sviluppatori, i quali cercano un ambiente di sviluppo senza avere gli oneri che derivano dalla gestione dell'hardware. L'utilizzo di una piattaforma PaaS offre agli sviluppatori la possibilità di sfruttare la scalabilità dinamica, ovvero è possibile ampliare o ridurre in modo flessibile le capacità richieste a seconda delle proprie esigenze, come accade con gli altri servizi cloud, l'automazione per i backup dei database e un set di linguaggi di programmazione specifici. Inoltre, consente di evitare le spese e le complessità legate all'acquisto e alla gestione di licenze software e alla realizzazione dell'infrastruttura. Il fatto che sia il provider ad occuparsi dell'infrastruttura è, al tempo stesso, un

vantaggio e uno svantaggio. Infatti, non è particolarmente utile quando l'applicazione deve essere portabile, quando vengono utilizzati linguaggi di programmazione proprietari o quando l'hardware e il software sottostanti devono essere personalizzati per migliorare le prestazioni dell'applicazione, proprio perché sono utilizzabili solo i linguaggi di programmazione e gli strumenti messi a disposizione dal Cloud Provider. Tale modello è tipicamente impiegato per analisi o business intelligence, per sviluppare o ampliare nuove interfacce di programmazione delle applicazioni (API) o applicazioni basate sul cloud. Inoltre, Platform as a Service può fungere da piattaforma di comunicazione, offrendo così contenuti audio e video o servizi di messaggistica istantanea. Esempi sono: Amazon Relational Database Service (RDS), Amazon DynamoDB, Amazon API Gateway, Google App Engine, Google Cloud SQL, Google Cloud Datastore.

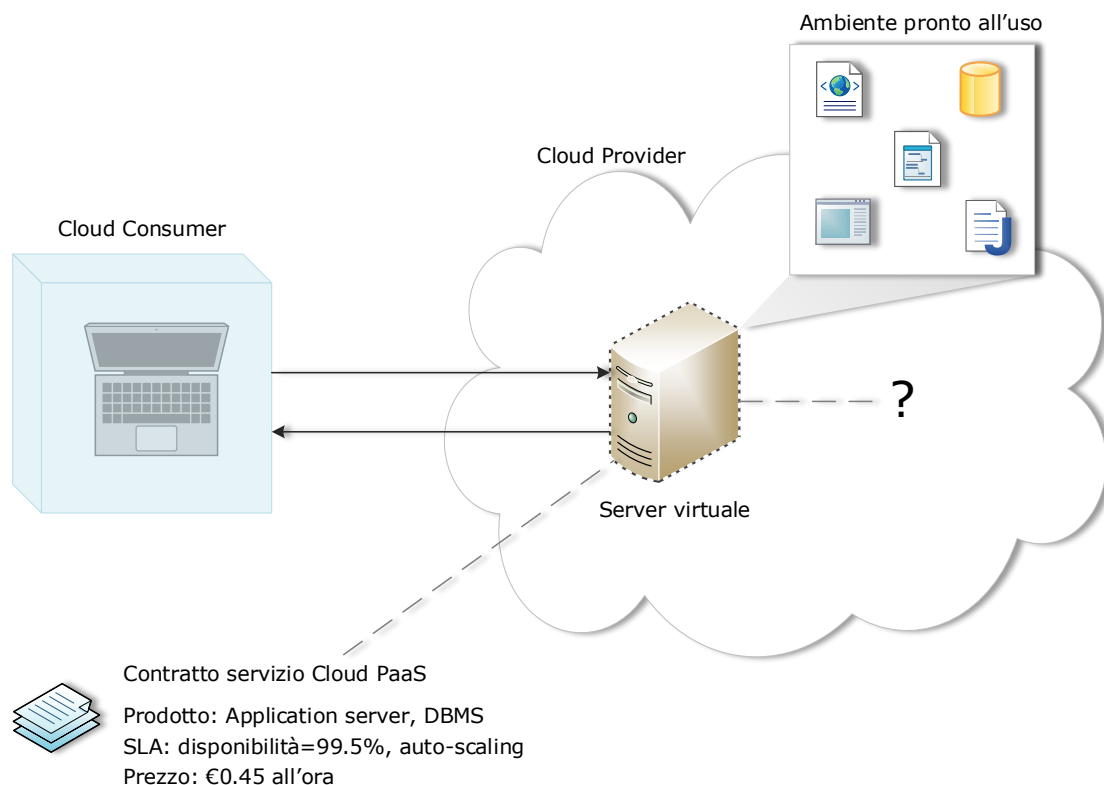


Figura 1.4 – Un Cloud Consumer accede ad un ambiente PaaS pronto all'uso. Il punto interrogativo indica che il consumatore è intenzionalmente protetto dai dettagli di implementazione della piattaforma.

- *Software as a Service (SaaS)*: consente agli utenti di connettersi, tramite Internet, ad applicazioni basate sul cloud. Il Cloud Provider gestisce l'ambiente cloud che ospita il software. Le applicazioni SaaS sfruttano l'architettura multi-tenant per utilizzare i pool di risorse. Degli aggiornamenti, della correzione dei bug e di altre attività generiche di manutenzione del software se ne occupa il provider SaaS. Gli utenti interagiscono con il software tramite un browser installato nei propri computer o dispositivi portatili, oppure utilizzano API per connettere il software ad altre funzioni. Il consumatore non gestisce o controlla né l'infrastruttura cloud sottostante, compresi rete, server, sistemi operativi, memoria, né le capacità delle singole applicazioni, con la possibile eccezione di limitate configurazioni a lui destinate. L'utente finale non ha bisogno di nessuna conoscenza informatica per utilizzare l'applicazione o i servizi erogati. Il principale vantaggio è proprio la possibilità di usare i servizi su qualsiasi dispositivo e in qualsiasi luogo. Questa tipologia di servizi viene spesso erogata gratuitamente oppure può richiedere una sottoscrizione basata sul tempo di utilizzo o sul numero di utenze. L'utilizzo del modello SaaS tendenzialmente riduce i costi derivanti dalle licenze, dalla gestione e installazione degli aggiornamenti. Tale modello non è adatto per applicazioni che richiedono una risposta in tempo reale oppure quelle in cui i dati vengono archiviati all'esterno del cloud. Tipicamente è impiegato per servizi di posta elettronica, applicazioni di project management, sistemi di gestione dei contenuti (CMS), programmi per la contabilità, file management, e-commerce, Customer Relationship Management (CRM), gestione degli archivi e pianificazione delle risorse umane.

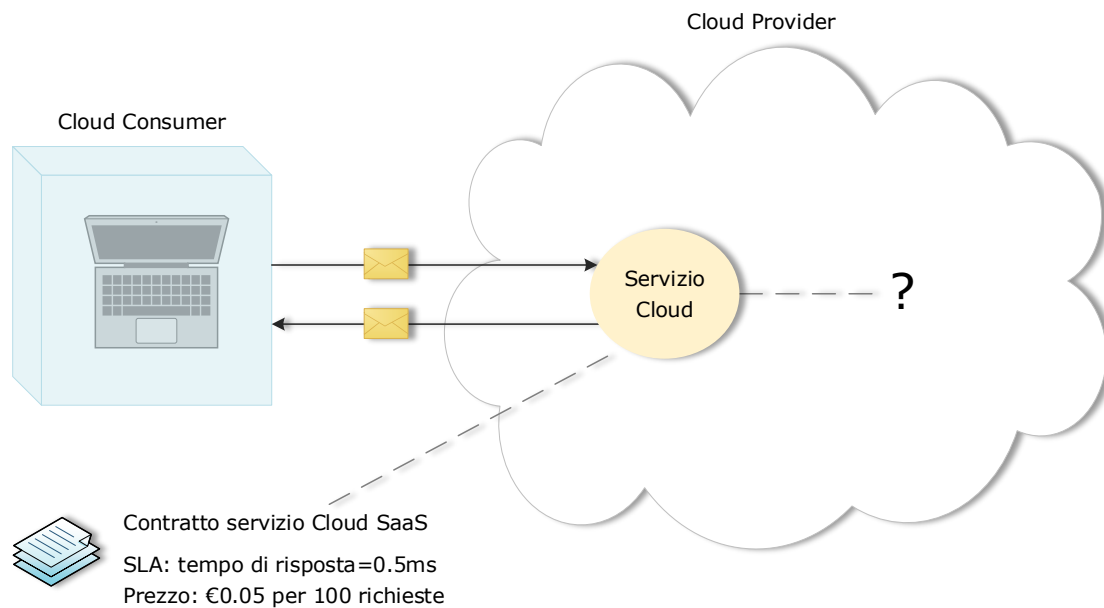


Figura 1.5 – Il Cloud Consumer ha accesso al contratto del servizio cloud, ma non ai dettagli di implementazione dell'infrastruttura sottostante.

Oltre alle tre tipologie descritte, ormai presenti e consolidate da diverso tempo, è importante citare altri due modelli ampiamente impiegati:

- *Function as a Service (FaaS)*: diffuso per la prima volta da un grande provider con il progetto Amazon AWS Lambda, seguito poi dai vari Google Cloud Functions e Microsoft Azure Functions, è sicuramente la tecnologia maggiormente impiegata nella realizzazione di applicazioni basate sul paradigma *serverless computing*. Con l'introduzione della tipologia FaaS, prendendo spunto da molti aspetti già offerti dal modello PaaS, si effettua un passo avanti in termini di astrazione, scalabilità e riduzione dei costi. Questo modello consente agli sviluppatori di creare, eseguire e gestire i pacchetti applicativi come funzioni, senza doversi occupare della propria infrastruttura, favorendo l'implementazione di un'architettura basata su *microservizi*. L'infrastruttura FaaS viene in genere utilizzata on demand, principalmente mediante un modello di esecuzione basato su eventi (*event-driven*), il quale permette di evitare l'esecuzione costante dei processi server in background, come accade con il modello PaaS. Un Cloud Provider rende la funzione disponibile e ne gestisce l'allocazione delle risorse. Essendo basate sugli eventi e non sulle risorse, le funzioni sono facilmente scalabili. Con il modello FaaS non è necessario pagare un canone per

disporre di server, storage o di un application server in cloud, ma la scalabilità dinamica delle risorse permette l'adozione del modello pay-per-use, con il quale i provider addebitano solo il costo di utilizzo delle risorse, quando ciò avviene, escludendo i tempi morti. Tale modello è perfetto per transazioni con volumi elevati, carichi di lavoro con frequenza sporadica come la generazione di report, l'elaborazione di immagini o attività programmate. Esempi di utilizzo comuni sono l'elaborazione dei dati, i servizi IoT, le app mobile o web.

- *Backend as a Service (BaaS)*: spesso accostato al modello IaaS, consente agli sviluppatori di esternalizzare gran parte degli aspetti del back end di un'applicazione web o mobile, permettendo loro di concentrarsi esclusivamente sulla scrittura del codice del front end di questa. Applicazioni web o mobile richiedono un elevato numero di funzionalità lato server, come la gestione del database, l'autenticazione degli utenti, le notifiche push e il cloud storage. Ognuno di questi servizi ha le proprie API, che devono essere incorporate singolarmente in un'app, un processo complicato che può richiedere molto tempo per gli sviluppatori di applicazioni. I provider di servizi BaaS forniscono un ponte tra il front end di un'applicazione e vari cloud-based back end, accessibile tramite kit di sviluppo software (SDK) o API. Essendo "plug-and-play", l'integrazione di tali servizi nei sistemi di un'azienda è estremamente più semplice e, talvolta, più affidabile, rispetto a svilupparli internamente.

Questi due modelli presentano differenze significative, nonostante spesso vengano erroneamente ritenuti molto simili. Mentre FaaS sono le applicazioni che vengono utilizzate per gestire e implementare i microservizi in modo più efficace, progettate come composizione di più funzioni interagenti ad eventi, nel modello BaaS troviamo servizi in esecuzione in background per gestire il back end dell'applicazione, realizzati come il provider preferisce e, gli sviluppatori, non devono neanche conoscerne la struttura. Pertanto, con le FaaS viene eliminata la necessità di mantenere un'infrastruttura always-on, infatti, essendo event-driven, vengono attivate solo quando necessarie, a differenza dei servizi BaaS, i quali richiedono molte più risorse server.

Infine, i servizi BaaS non sono progettati per scalare automaticamente a fronte di carichi elevati, aspetto invece cruciale in tutte le piattaforme FaaS, le quali sono scalabili automaticamente e forniscono quindi un livello di semplicità più elevato.

1.2 Modelli di distribuzione

La distribuzione delle risorse nel cloud è disponibile in diverse opzioni che permettono una categorizzazione dipendente dalle esigenze aziendali. I modelli di distribuzione dei servizi cloud riguardano principalmente i data center in cui sono installati tali servizi. Infatti, i servizi possono essere installati nei data center di un provider che li rende accessibili a tutti gli utenti, nei data center degli utenti stessi o su macchine riservate, situate però in data center di fornitori pubblici.

La definizione del cloud computing elaborata dal NIST delinea quattro modelli principali:

- *Private cloud*: l'infrastruttura cloud è fornita per uso esclusivo da parte di una singola organizzazione comprendente molteplici consumatori (ad esempio filiali). Può essere posseduta e gestita dall'organizzazione stessa, da una società terza o da una combinazione delle due e, può esistere all'interno o all'esterno delle proprie sedi. Il principale vantaggio di un cloud privato è che i servizi vengono forniti da elaboratori che si trovano nel dominio dell'utente e, quindi, l'utente ha il pieno controllo delle macchine sulle quali vengono conservati i dati e vengono eseguiti i suoi processi. In particolare, l'utente può applicare su queste macchine le politiche di sicurezza che ritiene più opportune per la protezione dei suoi dati. Una simile soluzione ha però alcuni svantaggi importanti da tenere in considerazione, infatti, la gestione di un cloud privato comporta costi di personale, amministrazione e manutenzione paragonabili a quelli di un normale data center, con conseguente allocazione più limitata delle risorse e quindi una scalabilità meno elastica. Infine, va tenuto in considerazione che un cloud privato si scontra con la diffusione dei dispositivi mobili, nel senso che un utente in movimento potrebbe avere un accesso limitato al proprio cloud privato, non potendo rispettare le misure di sicurezza previste ovunque egli si trovi.

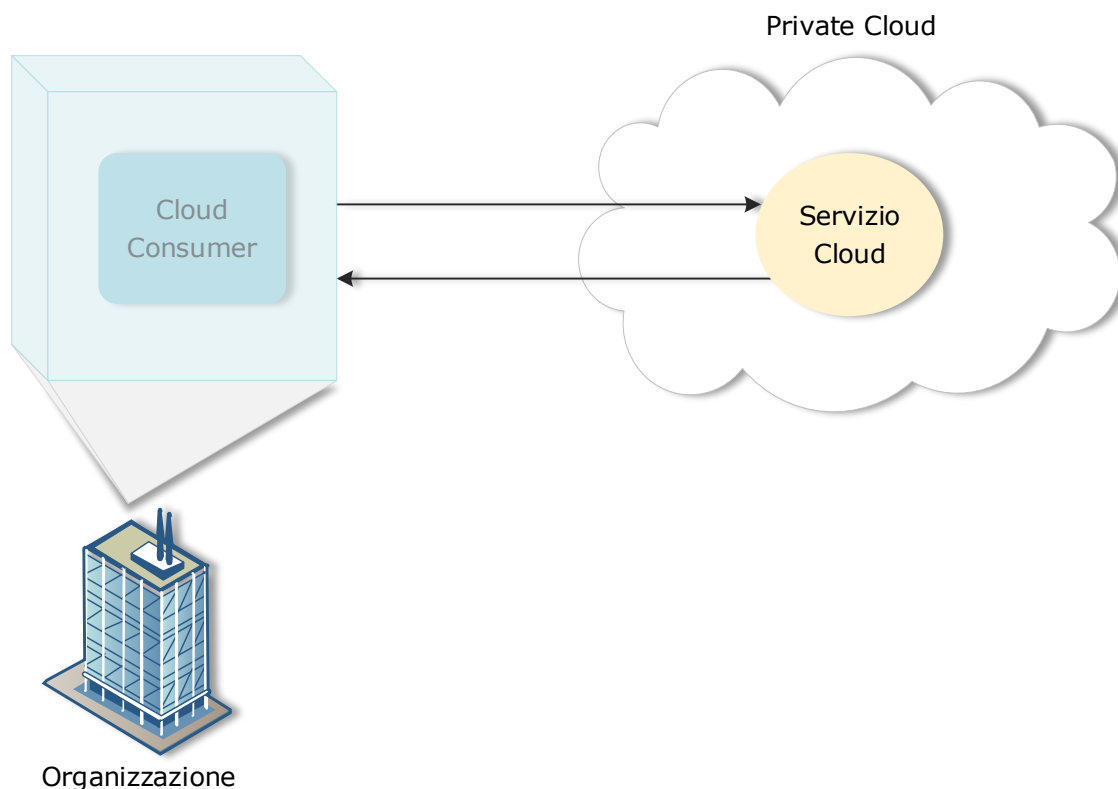


Figura 1.6 – Un Cloud Consumer accede a un servizio cloud ospitato sul cloud privato dell'organizzazione a cui appartiene tramite una VPN.

- *Public cloud*: l'infrastruttura cloud è fornita per un uso aperto a qualsiasi consumatore. Può essere posseduta e gestita da un'azienda, da un'organizzazione accademica, governativa oppure da una combinazione delle precedenti e, esiste all'interno delle sedi del Cloud Provider. Uno dei maggiori vantaggi del cloud pubblico per il consumatore consiste nel fatto che egli può richiedere l'utilizzo dei servizi cloud di cui necessita nel momento in cui effettivamente ne ha bisogno e, solo per il tempo in cui sono necessari. In questo modo, il cliente può ridurre gli investimenti in infrastrutture IT e ottimizzare l'utilizzo delle risorse interne, riuscendo a risolvere i picchi di calcolo periodici o imprevisti grazie all'ottima scalabilità dell'infrastruttura e all'elevata disponibilità di risorse. Un aspetto negativo, invece, è che il cliente non sempre ha il completo controllo dei suoi dati e dei suoi processi, quando questi vengono gestiti dai servizi cloud pubblici. Inoltre, il consumatore non sempre può definire una propria politica di sicurezza, ma deve spesso accettare quella dichiarata dal provider.

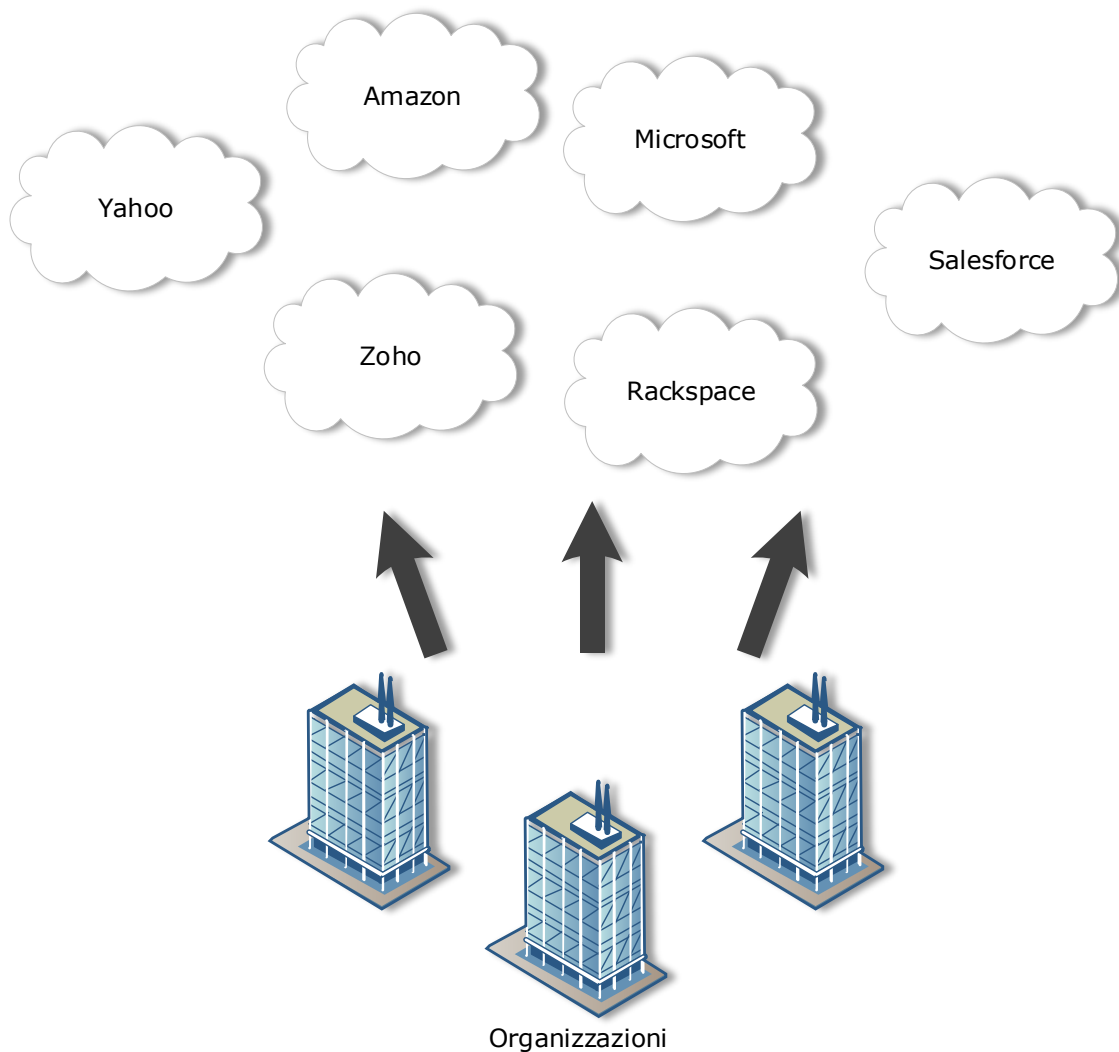


Figura 1.7 – Le organizzazioni agiscono come i Cloud Consumer quando accedono ai servizi cloud e alle risorse IT messe a disposizione da diversi Cloud Provider.

- *Community cloud*: l'infrastruttura cloud è fornita per uso esclusivo da parte di una comunità di consumatori appartenenti ad organizzazioni con interessi comuni e che hanno le stesse esigenze, come, ad esempio, potrebbero essere i vari soggetti della pubblica amministrazione. Può essere posseduta e gestita da una o più organizzazioni della comunità, da una società terza o una combinazione delle due e, può esistere all'interno o all'esterno delle proprie sedi.

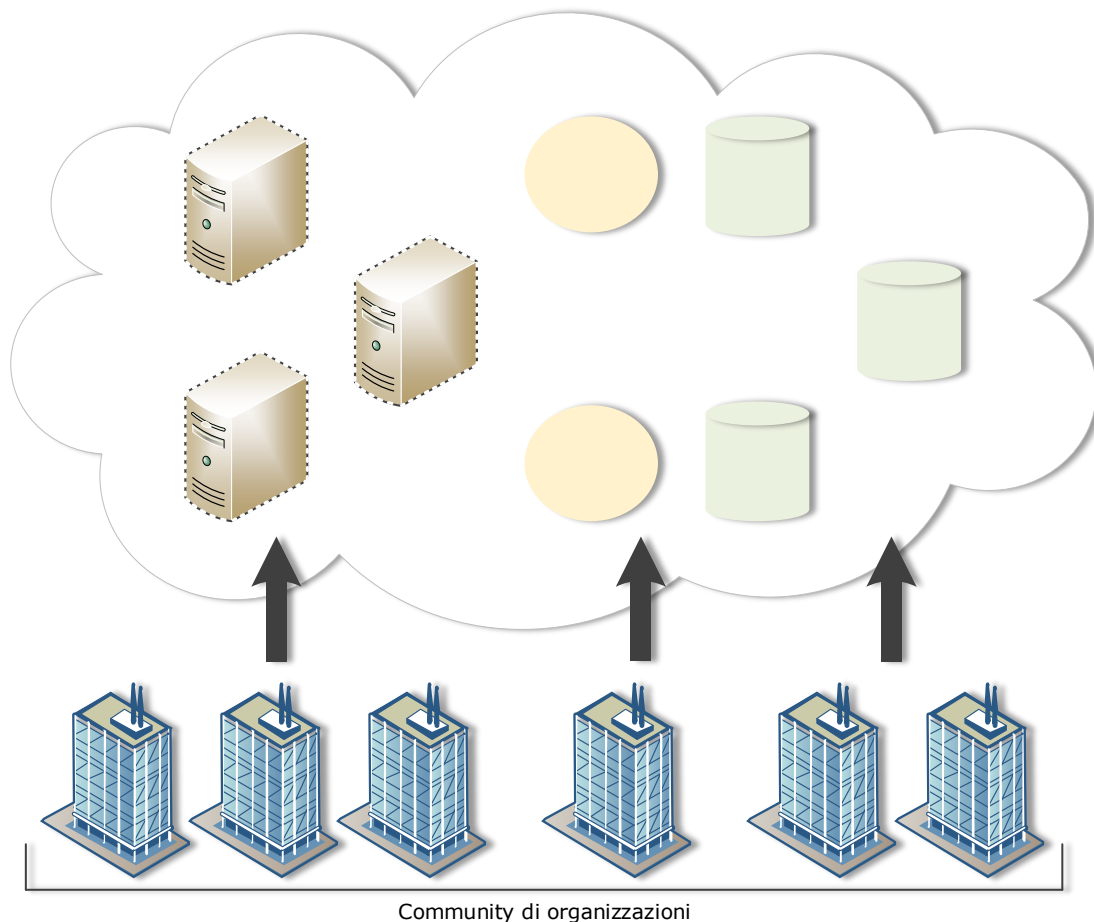


Figura 1.8 – Un esempio di “community” di organizzazioni che accedono alle risorse IT da un community cloud.

- *Hybrid cloud*: l’infrastruttura cloud è una composizione di due o più infrastrutture cloud (private, public o community) che rimangono entità distinte, ma sono unite attraverso tecnologie standard o proprietarie, che abilitano la portabilità di dati e applicazioni (ad esempio per bilanciare il carico di lavoro tra cloud). Il cloud ibrido può essere utilizzato con successo in vari casi. Ad esempio, un utente che dispone di un cloud privato, può utilizzare le risorse di un cloud pubblico per gestire improvvisi picchi di lavoro, che non possono essere soddisfatti facendo ricorso unicamente alle risorse disponibili nel cloud privato. Questa soluzione è facilmente implementabile quando il cloud privato è installato nello stesso data center del provider di servizi cloud pubblici. La differenza, in termini di efficienza dei costi, è che l’hybrid cloud fornisce una soluzione intermedia tra i due estremi, ossia tra le massime economie di scala ottenibili con l’adozione del public cloud e quelle più contenute, raggiungibili applicando il modello del private cloud. Basti ricordare che, a livello di efficienza dei

costi, i cloud pubblici possono fornire economie di scala maggiori rispetto ai cloud privati, facendo leva, per esempio, sulla gestione centralizzata delle risorse IT da parte del Cloud Provider. Il modello del cloud ibrido, in sostanza, permette di estendere questi vantaggi di costi a quante più funzioni possibile, affidandosi comunque al private cloud quando occorre proteggere con la massima sicurezza applicazioni e dati sensibili. In ogni caso, va tenuto presente che avere un cloud ibrido non significa avere un po' di cloud privato e un po' di cloud pubblico. La chiave di volta è l'integrazione, il che significa poter amministrare e controllare ogni risorsa IT, applicazione, dato e workload in modo armonico, minimizzando i rischi e incrementando la produttività. Un aspetto negativo è sicuramente la necessità di un'elevata compatibilità e integrazione tra le due infrastrutture cloud, difficile da realizzare soprattutto per quanto concerne la parte pubblica, nella quale l'organizzazione ha uno scarso controllo degli ambienti. Proprio per questo motivo, è necessario introdurre complessità architetturale, aggiungendo strumenti e livelli di astrazione per permettere l'omogeneità tra i sistemi e la portabilità delle applicazioni.

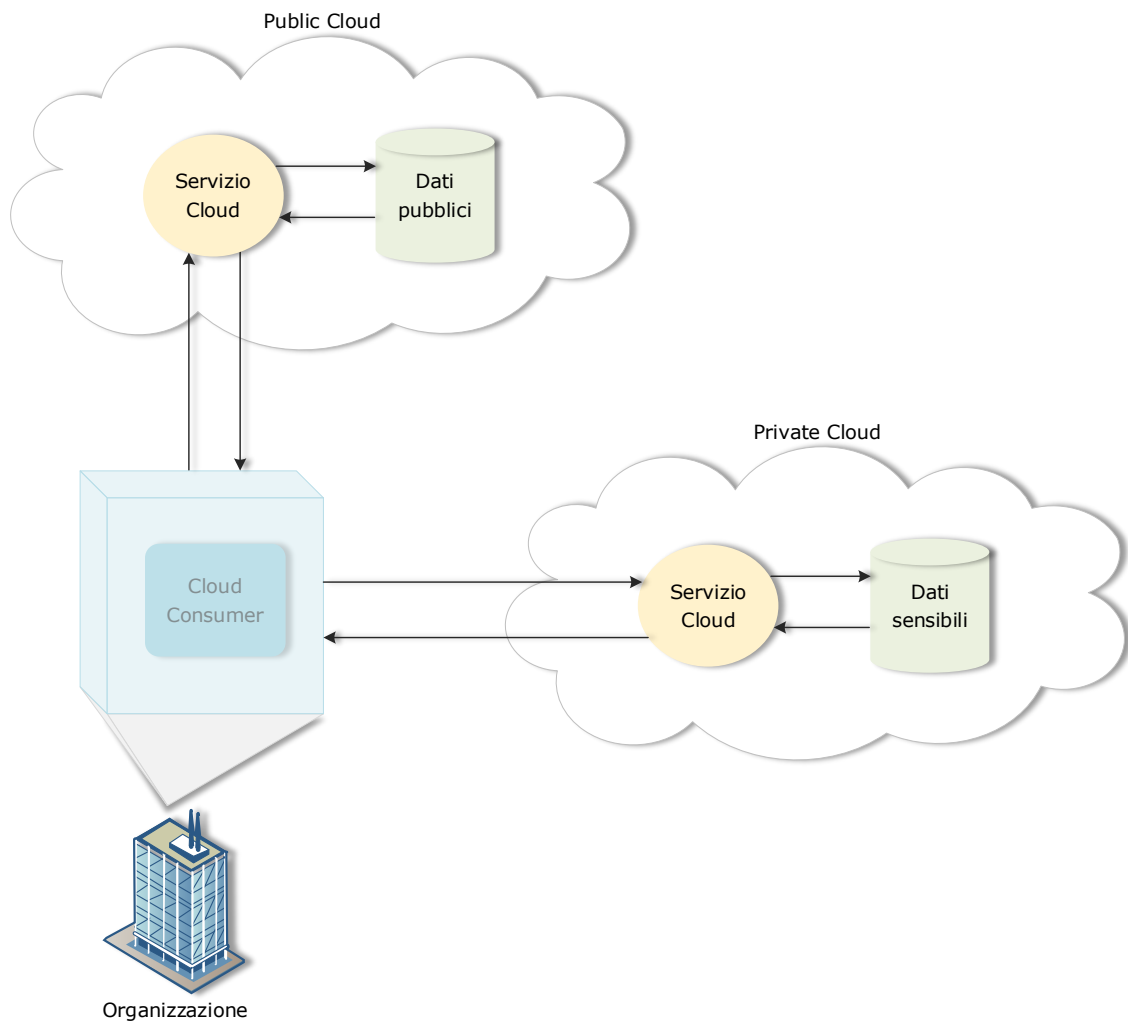


Figura 1.9 – Un'organizzazione utilizza un'architettura hybrid cloud, usufruendo sia di un cloud privato che di uno pubblico.

1.3 Modello concettuale e ruoli

Il modello concettuale di riferimento del cloud computing proposto dal NIST delinea cinque attori principali con i loro ruoli e responsabilità: cloud consumer, cloud provider, cloud auditor, cloud broker e cloud carrier. Ogni attore è un'entità (una persona o un'organizzazione) che partecipa a una transazione o processo e/o esegue attività nel cloud computing.

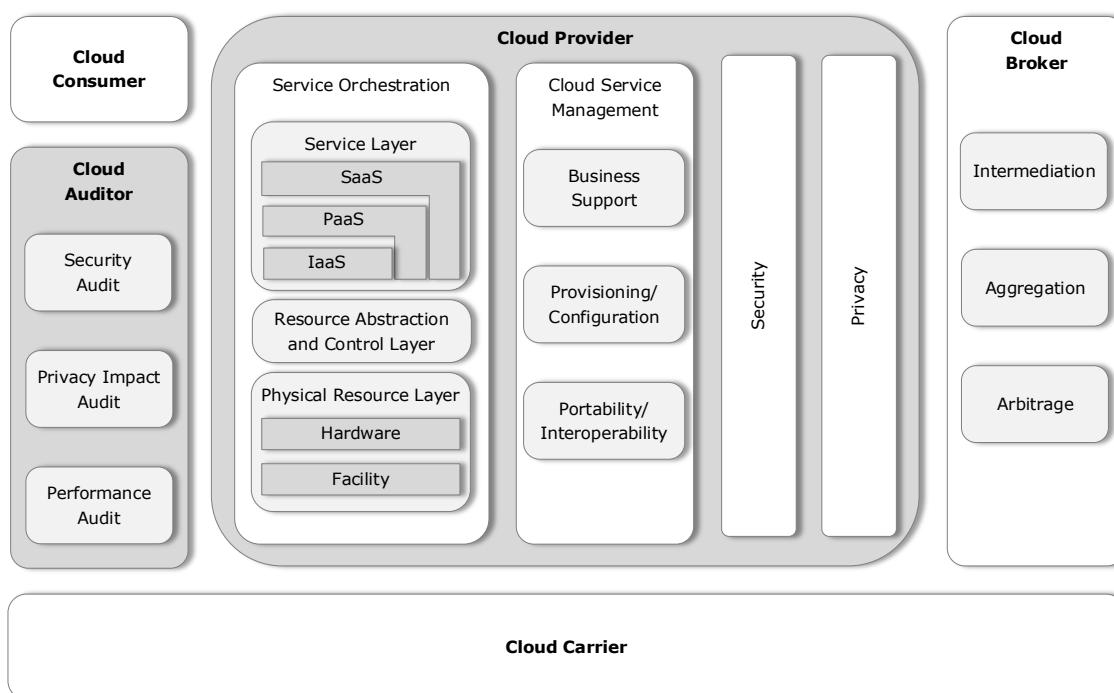


Figura 1.10 – Le entità coinvolte nel cloud computing secondo il modello concettuale di riferimento proposto dal NIST.

- *Cloud Provider* (o Cloud Service Provider, CSP): è l'entità responsabile di rendere il servizio utilizzabile alle parti terze interessate. Il Cloud Provider acquisisce e gestisce le infrastrutture elaborative necessarie a fornire i servizi, assicura l'esecuzione dei programmi che consentono i servizi, e le infrastrutture per erogare i servizi attraverso la rete. Le attività del CSP coprono cinque aree principali:
 - I. *Erogazione del servizio*: un'infrastruttura cloud può essere utilizzata secondo uno dei modelli di distribuzione precedentemente riportati.

- II. *Orchestrazione del servizio*: si riferisce all'organizzazione, al coordinamento e alla gestione dell'infrastruttura cloud per fornire le capacità di ottimizzazione dei servizi cloud.

Nel modello concettuale di riferimento proposto dal NIST viene identificato un framework a tre livelli per un sistema cloud generalizzato. Il livello superiore è il *Service Layer*, in cui un Cloud Provider definisce e fornisce ciascuno dei tre modelli di servizio. È qui che i consumatori utilizzano i servizi cloud attraverso le rispettive interfacce.

Il livello intermedio è il *Resource Abstraction and Control Layer*. Questo contiene i componenti di sistema che un Cloud Provider utilizza per fornire e gestire l'accesso alle risorse fisiche tramite l'astrazione software. Tale livello include tipicamente elementi software come hypervisor, macchine virtuali, archiviazione di dati virtuali e altri componenti di gestione e astrazione delle risorse, necessari per garantire un utilizzo efficiente, sicuro e affidabile.

Il livello più basso nel framework è il *Physical Resource Layer*, che include tutte le risorse di elaborazione fisiche, ovvero risorse hardware, come computer (CPU e memoria), reti (router, firewall, switch, collegamenti di rete e interfacce), componenti di archiviazione (dischi rigidi). Comprende anche risorse per il riscaldamento, la ventilazione, l'alimentazione, le comunicazioni e altri elementi dell'infrastruttura informatica fisica.

- III. *Gestione dei servizi cloud*: include tutte le funzioni necessarie per la gestione e il funzionamento dei servizi richiesti o proposti ai Cloud Consumer. Può essere descritta dal punto di vista del supporto aziendale, del provisioning e della configurazione e, dei requisiti di portabilità e interoperabilità.
- IV. *Sicurezza*: copre tutti i livelli dell'architettura di riferimento e, in generale, la responsabilità è condivisa tra Cloud Provider e Cloud Consumer. I primi dovrebbero garantire che la struttura che ospita i servizi cloud sia sicura e che il personale disponga di adeguati controlli in background, mentre, i Cloud Consumer controllano che l'offerta cloud soddisfi i requisiti di sicurezza e i criteri di conformità.

È anche importante notare che i criteri di conformità dipendono dalla giurisdizione legale del Paese in cui vengono forniti i servizi cloud e, possono variare da Paese a Paese.

- V. *Privacy*: i Cloud Provider dovrebbero proteggere la raccolta, l'elaborazione, la comunicazione e l'utilizzo sicuro, corretto e coerente delle informazioni personali.
- *Cloud Consumer*: rappresenta un individuo o un'organizzazione che mantiene una relazione commerciale e utilizza i servizi di cloud computing.
- Il consumatore esamina il catalogo dei servizi di un Cloud Provider, richiede specifici servizi e li utilizza.
- Il Cloud Consumer sottoscrive con il provider un contratto, il quale stabilisce gli accordi sui livelli di servizio (Service Level Agreement, SLA) per specificare i requisiti sulle prestazioni tecniche, che devono essere soddisfatti. Un Cloud Provider potrebbe anche elencare negli SLA un insieme di restrizioni e obblighi che il consumatore deve accettare.
- *Cloud Auditor*: si occupa di condurre una valutazione indipendente delle prestazioni e della sicurezza dei servizi cloud. Può valutare i servizi forniti da un Cloud Provider in termini di controlli di sicurezza, impatto sulla privacy, prestazioni e rispetto dei parametri del contratto sul livello di servizio.
- *Cloud Broker*: l'integrazione dei servizi cloud può risultare un'attività complessa da condurre per il Cloud Consumer, specie in un ambiente in forte evoluzione come quello del cloud computing. Invece di contattare direttamente il Cloud Provider, il consumatore può pertanto richiedere i servizi cloud attraverso un Cloud Broker. Quest'ultimo è il soggetto che gestisce l'impiego, le prestazioni, l'erogazione dei servizi cloud e cura le relazioni tra il Cloud Provider e il Cloud Consumer.

In generale, opera in tre aree:

- I. *Intermediation*: estende un servizio cloud fornendo servizi a valore aggiunto ai Cloud Consumer, come ad esempio la gestione dell'accesso, dell'identità o della sicurezza.
- II. *Aggregation*: combina e integra più servizi diversi in uno nuovo, assicurando l'integrazione e la sicurezza dei dati trasferiti tra il consumatore e i differenti provider.

- III. *Arbitrage*: è simile all'aggregazione dei servizi, tranne per il fatto che il broker ha maggiore flessibilità nella scelta dei servizi da più provider diversi, sulla base di criteri di economicità o disponibilità.
- *Cloud Carrier*: agisce come un intermediario, fornendo la connettività ed il trasporto di servizi cloud tra il Cloud Consumer e il Cloud Provider. Il Cloud Carrier fornisce al consumatore l'accesso attraverso le reti e i dispositivi di accesso. Per esempio, i Cloud Consumer possono ottenere servizi cloud attraverso i dispositivi di accesso alla rete, come computer desktop, computer portatili, smartphone e altri dispositivi mobili. La distribuzione dei servizi cloud è normalmente fornita dagli operatori di rete e di telecomunicazione. Si noti che un Cloud Provider stabilirà accordi inerenti al livello di servizio con un Cloud Carrier per fornire servizi coerenti con il livello di SLA offerto ai consumatori e, potrebbe richiedere delle connessioni dedicate e crittografate con i Cloud Consumer.

1.4 Il ruolo della virtualizzazione

I server di un data center sono macchine molto potenti e nessuna singola applicazione può sfruttarne la loro potenza di calcolo in maniera continuativa ed esclusiva per lunghi periodi di tempo, ma è opportuno dividerne le risorse tra molti utenti.

La *virtualizzazione* crea uno strato di astrazione tra le applicazioni e l'hardware sottostante, permettendo di condividere i server e lo storage, aumentarne radicalmente il tasso di utilizzo e di spostare facilmente le applicazioni da un server fisico ad un altro. L'obiettivo è appunto quello di supportare la portabilità, migliorare l'efficienza, aumentare l'affidabilità e proteggere l'utente dalla complessità del sistema.

Benché la virtualizzazione risalga agli anni '60, si è diffusa soltanto all'inizio degli anni 2000. Le tecnologie che hanno consentito la virtualizzazione, come gli hypervisor, sono state sviluppate decenni fa, al fine di fornire a più utenti l'accesso simultaneo a computer con un'elaborazione batch attiva. Molte aziende hanno adottato l'elaborazione batch come modello di calcolo, poiché consentiva di eseguire attività di routine, come la gestione dei libri paga, migliaia di volte più rapidamente.

Tuttavia, nel corso dei decenni successivi, il problema della presenza di più utenti su un unico computer è stato affrontato scegliendo soluzioni diverse dalla virtualizzazione.

Negli anni '90, gran parte delle aziende utilizzava server fisici associati a un singolo provider, che non consentivano l'esecuzione delle applicazioni di altri provider. Le aziende hanno iniziato, così, ad aggiornare i propri ambienti IT con "commodity server", sistemi operativi e applicazioni meno costosi di vari provider.

È a questo punto che la virtualizzazione ha iniziato a diffondersi. Era la soluzione naturale a due problemi: permetteva alle aziende di ottenere partizioni sui propri server e, di eseguire le app esistenti su più tipologie e versioni di sistemi operativi. L'utilizzo dei server è stato reso più efficiente, riducendo così i costi associati all'acquisto, alla configurazione, al raffreddamento e alla manutenzione.

L'ampia applicabilità della virtualizzazione ha contribuito a ridurre il vendor lock-in e ha posto le basi del cloud computing. Oggi è utilizzata da moltissime aziende ed è spesso necessario disporre di software specifici per la gestione della virtualizzazione, i quali consentono di monitorare l'ambiente.

I software definiti *hypervisor*, anche conosciuti come *Virtual Machine Monitor (VMM)*, separano le risorse fisiche dagli ambienti virtuali che le richiedono. Possono essere eseguiti in un sistema operativo oppure installati direttamente su un hardware (server), che rappresenta la modalità di virtualizzazione utilizzata da gran parte delle aziende.

Gli hypervisor ripartiscono le risorse fisiche in modo che gli ambienti virtuali possano utilizzarle. A seconda delle esigenze, le risorse vengono suddivise e trasferite dall'ambiente fisico ai vari ambienti virtuali. Gli utenti interagiscono ed eseguono calcoli all'interno dell'ambiente virtuale (in genere definito macchina guest o macchina virtuale). La macchina virtuale funziona come un unico file dati, che, a differenza di qualsiasi file digitale, è utilizzabile anche nel momento in cui viene spostato da un computer a un altro o aperto su più computer.

L'hypervisor svolge un ruolo simile a un sistema operativo (SO), gestendo varie VM, proprio come il SO gestisce i processi dell'utente. L'unica differenza è che ogni VM può appartenere a un utente diverso e indipendente e, può contenere il proprio sistema operativo, consentendo, ad esempio, a Windows e Linux di coesistere sulla stessa piattaforma hardware.

Quando l'ambiente virtuale è in funzione e un utente o un programma esegue un'istruzione che richiede risorse aggiuntive dall'ambiente fisico, l'hypervisor riporta la richiesta sul sistema fisico e memorizza nella cache le modifiche, le quali vengono applicate quasi in tempo reale.

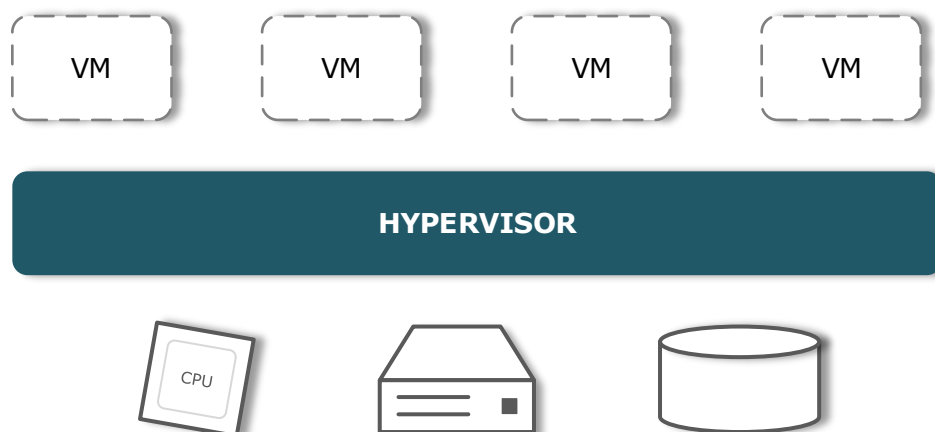


Figura 1.11 – Diverse macchine virtuali condividono le stesse risorse fisiche.

L'esecuzione di più VM sullo stesso server consente alle applicazioni di condividere meglio le risorse del server e ottenere un maggiore utilizzo del processore. È probabile che le richieste istantanee di risorse delle applicazioni in esecuzione contemporaneamente siano diverse e si completino a vicenda, riducendo così al minimo il tempo di inattività del server.

La virtualizzazione è vantaggiosa sia per gli utenti che per i provider di servizi cloud. Gli utenti cloud apprezzano la virtualizzazione perché consente un migliore isolamento delle applicazioni l'una dall'altra rispetto al tradizionale modello di condivisione dei processi. I CSP godono di maggiori profitti grazie al basso costo per la fornitura di servizi cloud. Un altro vantaggio è che uno sviluppatore può scegliere di sviluppare l'applicazione in un ambiente familiare e, con un sistema operativo a sua scelta.

D'altra parte, la virtualizzazione contribuisce ad aumentare la complessità del software di sistema e ha effetti collaterali indesiderati sulle prestazioni e sulla sicurezza dell'applicazione.

1.5 Architettura a microservizi e container

Un ambiente cloud richiede una differente visione su come strutturare un'applicazione e su come questa operi in tale ambiente altamente distribuito. Le applicazioni abilitate per il cloud sono progettate per essere modulari, distribuite e gestite in modo automatizzato. Queste caratteristiche richiedono tecnologie che vadano oltre quelle impiegate per lo sviluppo dei software tradizionali.

È comune pensare di poter partire da un'applicazione *monolitica* esistente e spostarla semplicemente nel cloud. Sebbene sia fisicamente possibile, non si ottiene alcun vantaggio in termini di costi o flessibilità.

Le applicazioni monolitiche sono facili da implementare, poiché hanno una sola base di codice, tipicamente raccolte in un unico progetto e vengono distribuite all'interno di un unico pacchetto, spesso eseguito da un singolo host.

Nelle architetture monolitiche tutti i processi sono strettamente collegati tra loro e vengono eseguiti come un singolo servizio. Ciò significa che se un processo dell'applicazione sperimenta un picco nella richiesta, è necessario ridimensionare l'intera architettura. Aggiungere o migliorare una funzionalità dell'applicazione monolitica diventa più complesso, in quanto sarà necessario aumentare la base di codice. Un nuovo sviluppatore che entra nel team ha bisogno di imparare il funzionamento dell'intera applicazione, indipendentemente da quello che deve sviluppare e, ogni piccola modifica deve passare attraverso un test completo dell'intera applicazione prima di essere distribuita. Inoltre, l'unico modo per poter scalare un'applicazione monolitica è quello di replicare l'intera applicazione con conseguente aumento di costi e risorse necessarie. Tale complessità limita quindi la sperimentazione e, rende più difficile implementare nuove idee. Queste architetture rappresentano un ulteriore rischio per la disponibilità dell'applicazione, poiché la presenza di numerosi processi dipendenti e strettamente collegati aumenta l'impatto di un errore in un singolo processo.

Il modo più pratico per ottenere benefici dal cloud è pensare in modo modulare, ecco perché i *microservizi* sono così importanti per il cloud computing.

I microservizi sono un approccio architetturale allo sviluppo di applicazioni costituite da frammenti di codice indipendenti l'uno dall'altro e dalla piattaforma di sviluppo sottostante. Una volta creato, ogni microservizio esegue un processo univoco e comunica

tramite API ben definite e standardizzate. Questi servizi sono definiti in un catalogo in modo che gli sviluppatori possano individuare più facilmente il servizio giusto e comprendere la governance e le regole di utilizzo.

Suddividere un'applicazione nelle relative funzioni base ed evitare le insidie dell'approccio monolitico possono sembrare concetti familiari, perché lo stile architetturale dei microservizi è simile a quello dell'architettura *SOA (Service-Oriented Architecture)*, uno stile di progettazione del software ormai consolidato, in cui le app sono strutturate in servizi riutilizzabili che comunicano fra loro tramite un *Enterprise Service Bus (ESB)*. Nel complesso, questa suite di servizi integrati attraverso un ESB costituisce un'applicazione. Tuttavia, poiché l'ESB rappresenta un singolo punto di errore per l'intero sistema, potrebbe comportare un ostacolo per l'intera organizzazione.

I microservizi possono comunicare tra loro, in genere in modalità stateless, permettendo di realizzare app con una maggiore tolleranza agli errori e meno dipendenti da un singolo ESB. Con un'architettura basata su microservizi, un'applicazione è realizzata da componenti indipendenti che eseguono ciascun processo applicativo come un servizio. I servizi sono realizzati per le funzioni aziendali e ognuno di essi esegue una sola funzione. Poiché eseguito in modo indipendente, ciascun servizio può essere aggiornato, distribuito e ridimensionato per rispondere alla richiesta di funzioni specifiche di un'applicazione. Le architetture a microservizi permettono di scalare e sviluppare le applicazioni in modo più rapido e semplice, permettendo di promuovere l'innovazione e accelerare il time-to-market di nuove funzionalità.

Naturalmente nessuna architettura è esule da svantaggi e, in genere, ogni architettura applicativa che tenta di risolvere i problemi di scalabilità ha una serie di criticità da affrontare, data la natura complessa dei sistemi distribuiti. Il partizionamento di un'applicazione in servizi indipendenti implica l'esistenza di più parti da mantenere corrette e coerenti per tutto il ciclo di vita dell'applicazione e, quindi, di una più complessa orchestrazione. Per questo è necessario implementare un alto livello di automazione di tutti i processi.

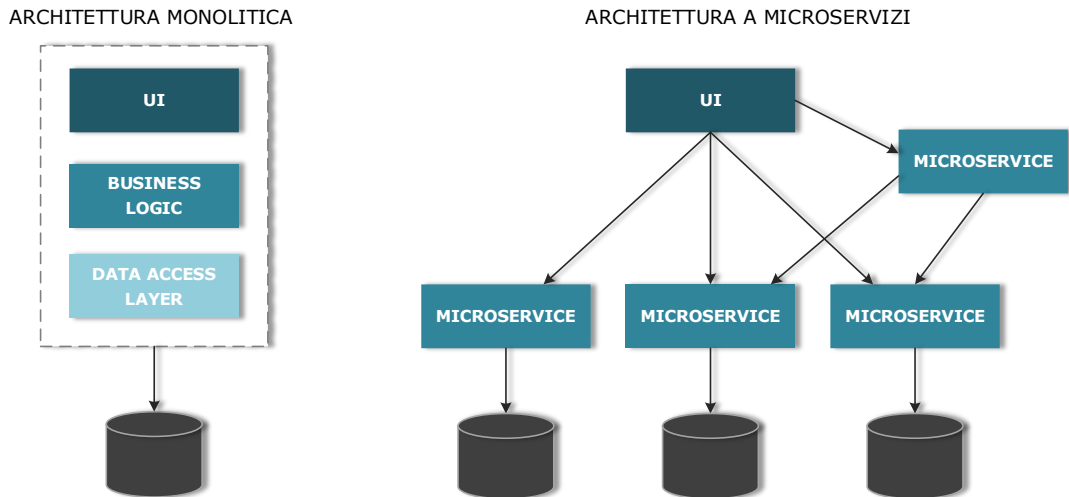


Figura 1.12 – Differenza tra un architettura monolitica ed una basata su microservizi.

I microservizi sono progettati per essere confezionati all'interno di *container*, i quali si basano sulla virtualizzazione a livello di sistema operativo piuttosto che sulla virtualizzazione dell'hardware. In parole povere, sfruttano le funzionalità del sistema operativo per isolare i processi e controllare l'accesso di questi a CPU e memoria, invece di richiedere l'esecuzione di un kernel virtualizzato, come avviene per le macchine virtuali. Ciò permette di ospitare un considerevole numero di container sulla stessa macchina fisica. Un'applicazione in esecuzione all'interno di un container viene isolata da un'altra in esecuzione in un container diverso ed entrambe sono isolate dal sistema fisico in cui vengono eseguite. I container sono portabili e le risorse utilizzate possono essere limitate.

Prima dell'introduzione dei container, era già comune la pratica di virtualizzare le risorse necessarie a un dato software. Le consolidate tecnologie di virtualizzazione (virtual machine o VM) offrono certamente un buon grado di isolamento, ma, al tempo stesso, conducono a un elevato consumo di risorse di calcolo, poiché ogni VM obbliga l'esecuzione dell'intero sistema operativo e di tutte le librerie necessarie al suo ambiente. Ciò impone un limite al numero di VM che uno stesso host può ospitare, se si vogliono preservare le prestazioni generali del sistema. Inoltre, le VM risultano essere poco portabili e le operazioni di migrazione richiedono spesso interventi manuali non esenti da rischi ed errori.

I container esistono da decenni, ma è cresciuta in maniera esponenziale nel 2013 con l'introduzione di *Docker*, un progetto open source nato proprio con lo scopo di

automatizzare la distribuzione di applicazioni sotto forma di container leggeri, portabili e autosufficienti, eseguibili sia su cloud che in locale, a sostituzione delle VM.

Si può pensare ad un container come l'insieme dei dati necessari all'esecuzione di un'applicazione, quali, ad esempio, librerie, altri file eseguibili, file system, file di configurazione e script, impacchettati in modo che questa possa essere eseguita in modo coerente su cloud e in locale. Questo incapsulamento del codice è importante per gli sviluppatori, i quali non devono sviluppare codice basato su ogni singolo ambiente.

Quindi, con la containerizzazione, uno sviluppatore può trasferire codice da un desktop a una macchina virtuale (VM) o da un sistema operativo Windows a Linux in esecuzione su un cloud pubblico, con la certezza che il codice venga eseguito in modo prevedibile. Lo sviluppatore può semplicemente concentrarsi sulla logica dell'applicazione e sui servizi richiesti per essere supportata.

Containerizzare applicazioni e microservizi necessita di un modo per gestire i container, i quali possono essere anche centinaia o addirittura migliaia in una singola applicazione. Le piattaforme di orchestrazione dei container aiutano a semplificare i processi come l'installazione, la scalabilità e la gestione delle applicazioni containerizzate. Oltre a ciò, si occupano anche di garantire una certa tolleranza ai guasti, monitorando l'esecuzione dei container, per poter sostituire eventuali entità non funzionanti. *Kubernetes* di Google e *Docker Swarm* sono le piattaforme più popolari di orchestrazione dei container.

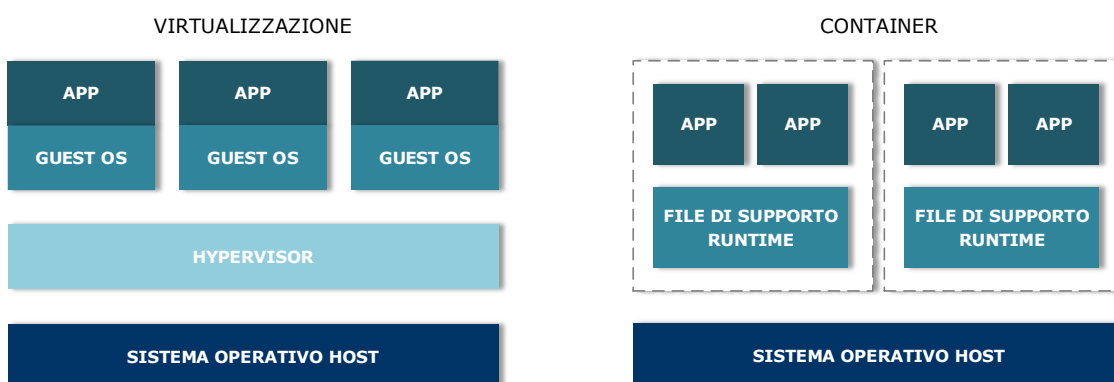


Figura 1.13 – Differenza tra virtualizzazione e containerizzazione.

1.6 Serverless computing

Il *serverless computing* è un modello di sviluppo cloud native che consente agli sviluppatori di creare ed eseguire applicazioni senza gestire i server.

Anche se in questo modello i server vengono comunque utilizzati, sono astratti dallo sviluppo delle applicazioni. Gli autori del framework Serverless, un popolare strumento grazie al quale è possibile realizzare applicazioni serverless, sottolineano questa importante caratteristica, ribadendo che:

*“Così come l’internet wireless ha dei cavi da qualche parte, le architetture serverless continuano ad avere dei server da qualche parte. Quello che ‘serverless’ vuol dire davvero è che, in quanto sviluppatore, non bisogna pensare a questi server, focalizzandosi solo sul codice.”*³

Il modello serverless è diverso dagli altri modelli di cloud computing, poiché il Cloud Provider è responsabile di gestire sia l’infrastruttura cloud che la scalabilità delle applicazioni, le quali vengono distribuite in container che vengono avviati on demand al momento della chiamata.

In un modello di cloud computing IaaS standard, gli utenti acquistano unità di capacità, ovvero pagano a un provider di cloud pubblico i componenti server che sono utilizzati per l’esecuzione delle proprie applicazioni, i quali sono sempre attivi, anche quando non necessari. L’utente è responsabile di aumentare la capacità del server durante i picchi di domanda e ridurla quando non occorre più.

Con un’architettura serverless, invece, le applicazioni vengono avviate solo quando necessario. Quando un evento attiva l’esecuzione del codice, il provider di cloud pubblico assegna dinamicamente le risorse per tale codice e l’utente paga il servizio solo fino al termine dell’esecuzione. Il pagamento, dunque, è calcolato in base al solo tempo di esecuzione e, non in base alla quantità di risorse utilizzate.

Oltre ai vantaggi in termini di costo ed efficienza, l’elaborazione serverless evita agli sviluppatori le attività di routine manuali, necessarie per garantire la scalabilità delle applicazioni e il provisioning del server.

³ Serverless Framework. *Why Serverless?* [online].
Disponibile su <<https://www.serverless.com/learn/why/>> [Data di accesso: 23/02/2021].

Il modello serverless consente di affidare al Cloud Provider attività di routine quali, ad esempio, gestione del sistema operativo e file system, applicazione delle patch di sicurezza, bilanciamento del carico, gestione della capacità, gestione della scalabilità, registrazione e monitoraggio.

È possibile realizzare un'applicazione completamente serverless, oppure in parte serverless e in parte costituita da microservizi convenzionali.

In genere i prodotti di serverless computing rientrano in due categorie, BaaS e FaaS.

Paul Johnston, cofondatore della conferenza sulle tecnologie serverless dedicata agli sviluppatori, ServerlessDays, ex Senior Developer Advocate del team Serverless di AWS, oltre ad asserire che serverless è un approccio, non una tecnologia, propone un'interessante visione dal punto di vista economico, nella quale evidenzia come la scelta di un'architettura serverless sia dettata innanzitutto da un'esigenza commerciale:

*“Un'applicazione serverless è quella che fornisce il massimo valore aziendale per tutto il ciclo di vita dell'applicazione e che non ha costi se nessuno la utilizza, esclusi quelli di archiviazione dei dati.”*⁴

Johnston prosegue con il motivo per cui spesso viene commesso un errore a livello concettuale, identificando serverless come sinonimo di FaaS:

*“FaaS è la tecnologia abilitante più semplice e più ovvia per le applicazioni che utilizzano un approccio serverless.”*⁴, ma:

*“Serverless è molto più di un semplice FaaS.”*⁵

Il modello serverless, infatti, può essere utilizzato anche dai microservizi e dalle applicazioni tradizionali, a condizione che questi siano containerizzati e rispettino i requisiti di scalabilità dinamica e gestione dello stato.

Naturalmente sarebbe possibile unire, in una stessa applicazione, l'utilizzo delle FaaS e dei BaaS, in modo da avere un ambiente interamente serverless, ma con il rischio di limitare eccessivamente la flessibilità e la personalizzazione dei sistemi.

⁴ PAUL JOHNSTON, 2019. What is Serverless? The “2020” edition. [online]. Disponibile su <<https://medium.com/swlh/what-is-serverless-the-2020-edition-5a2f21581fe5>> [Data di accesso: 23/02/2021].

⁵ PAUL JOHNSTON, 2018. Serverless: It's much much more than FaaS. [online]. Disponibile su <<https://pauljohnston.medium.com/serverless-its-much-much-more-than-faas-a342541b982e>> [Data di accesso: 23/02/2021].

Inoltre, si potrebbe incorrere nel problema del *vendor lock-in*, il quale si verifica quando l'applicazione è fortemente dipendente dal fornitore di servizi cloud e, ciò rende complessa e costosa l'eventuale decisione di spostarsi su un altro provider.

L'elemento principale nell'invocazione di un processo serverless è l'*API Gateway*, uno strumento di gestione delle API che si colloca tra un client e una raccolta di servizi back end.

Un API Gateway è un server HTTP che intercetta tutte le richieste API dai client e le instrada al microservizio desiderato. In genere, gestisce una richiesta invocando più microservizi e aggregando i risultati se più utenti richiedono un servizio dallo stesso endpoint. Effettua una mappatura dei parametri di una richiesta HTTP in un input più conciso per la funzione FaaS, oppure, consente il passaggio dell'intera richiesta HTTP, solitamente come oggetto JSON. La funzione FaaS, una volta presa in carico la richiesta, la esegue e restituisce il risultato all'API Gateway, che a sua volta lo trasforma in una risposta HTTP da inoltrare al chiamante originale.

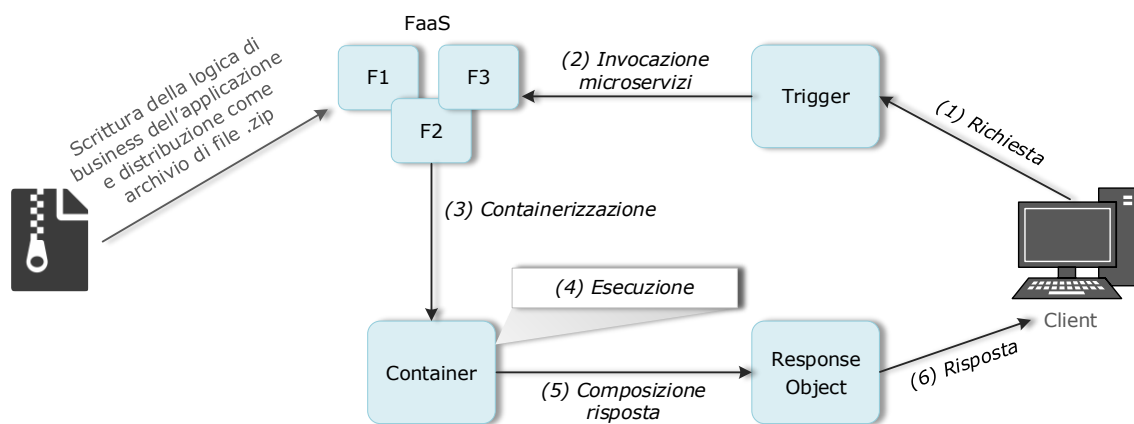


Figura 1.14 – Invocazione di un processo serverless.

Oltre alla funzione di instradamento delle richieste, gli API Gateway gestiscono anche attività comuni, come autenticazione dell'utente, validazione dell'input, limitazione di velocità ed elaborazione di statistiche.

Amazon Web Services ha il proprio API Gateway, chiamato appunto *Amazon API Gateway*, il quale è un servizio BaaS, completamente gestito, che consente agli sviluppatori di definire gli endpoint HTTP di un'API REST o di un'API WebSocket e connettere tali endpoint con la logica di business back end corrispondente.

Insieme ad AWS Lambda, Amazon API Gateway costituisce la parte rivolta all'applicazione dell'infrastruttura serverless di AWS.

Se da un lato il serverless computing può incrementare la produttività degli sviluppatori e ridurre i costi operativi, accelerando notevolmente il time-to-market dell'applicazione e stimolando una sperimentazione continua, dall'altro, il fatto di non eseguire i server internamente e di non controllare la logica lato server comporta alcuni svantaggi.

Con qualsiasi strategia di outsourcing viene ceduto il controllo di alcuni dei sistemi a un provider esterno. Tale mancanza di controllo può manifestarsi come tempi di inattività del sistema, limiti imprevisti, modifiche ai costi, perdita di funzionalità, aggiornamenti forzati delle API e altro ancora. Charity Majors, CEO e cofondatrice di Honeycomb, azienda specializzata nel supporto di team di ingegneri alla comprensione approfondita dei propri sistemi distribuiti attraverso l'osservabilità, delinea questo problema in modo molto più dettagliato, sottolineando che:

*“[Il servizio del provider], se è intelligente, imporrà forti vincoli su come utilizzarlo, in modo che sia più probabile raggiungere i propri obiettivi di affidabilità. Quando gli utenti hanno flessibilità e maggiore libertà di scelta, si crea caos e inaffidabilità. Se la piattaforma deve scegliere tra la tua felicità e quella di migliaia di altri clienti, sceglierà sempre i tanti invece di uno, come dovrebbe.”*⁶

La perdita di controllo può determinare il problema del *vendor lock-in*, oltre alle rilevanti tematiche relative alla sicurezza, la quale viene gestita attraverso i meccanismi offerti dal Cloud Provider. Questi, potrebbero non essere sufficienti a soddisfare i requisiti di sicurezza previsti per l'applicazione. Tuttavia, se da un lato il paradigma serverless appare poco gestibile dal punto di vista della sicurezza, dall'altro, bisogna considerare alcune caratteristiche che lo rendono, per certi aspetti, più sicuro:

- Il codice è eseguito su immagini di container ben consolidate e standardizzate, dunque, le vulnerabilità del sistema operativo e del kernel non sono significativamente rilevanti.

⁶ CHARITY MAJORS, 2016. *Operational Best Practices #serverless* [online]. Disponibile su <<https://charity.wtf/2016/05/31/operational-best-practices-serverless/>> [Data di accesso: 23/02/2021].

- La natura effimera e stateless dell'elaborazione serverless rende l'applicazione meno incline alle minacce. Infatti, le funzioni serverless, come nel caso di AWS Lambda, hanno un tempo di esecuzione relativamente breve, dopo il quale, i container che le ospitano vengono riciclati. Il fatto che le funzioni serverless siano temporanee e, pertanto, non hanno la necessità di mantenere il loro stato in memoria, riduce il rischio di attacchi a lungo termine.
 - L'architettura basata sui microservizi permette la definizione di policy per la concessione di privilegi minimi ad ogni funzione. In tal modo, è possibile ridurre in modo significativo la potenziale superficie di attacco.
- Inoltre, tramite l'API Gateway è possibile limitare il numero di richieste al secondo.

In un'applicazione non serverless, se la *latenza* tra i componenti dell'applicazione è un problema, questi possono generalmente essere ricollocati in modo ottimale (all'interno dello stesso rack o sulla stessa istanza host) oppure possono essere riuniti nello stesso processo. Inoltre, i canali di comunicazione tra i componenti possono essere ottimizzati per ridurre la latenza.

Contrariamente, questo rappresenta la maggiore limitazione delle applicazioni serverless. Infatti, se nelle tradizionali architetture *always on* i server sono sempre in esecuzione e in attesa delle richieste da elaborare, nelle piattaforme serverless, invece, i tempi di risposta di una funzione per la quale non esiste ancora nessuna istanza pronta a rispondere, risultano essere dilatati e possono variare da pochi millisecondi a diversi secondi.

Quando un utente invoca una funzione, questa viene tipicamente messa in esecuzione in un container, il quale, al termine delle operazioni, viene mantenuto attivo per un certo periodo di tempo limitato (stato di *warm*), per poi essere distrutto.

In tal modo, prima che il container venga distrutto, è possibile eseguire istantaneamente (*warm start*) una nuova richiesta per la stessa funzione.

Il problema del *cold start* si verifica quando occorre gestire una richiesta e non sono disponibili container in stato di *warm* già pronti a servirla. Tipicamente, ciò accade nel momento in cui una funzione viene invocata per la prima volta, dopo averne modificato la configurazione, in seguito ad un ridimensionamento (in particolare in caso di più istanze in esecuzione contemporaneamente) oppure semplicemente quando la funzione non viene richiamata da un po' di tempo. Questo implica l'aggiunta di un certo ritardo

nella gestione della richiesta, necessario per istanziare e inizializzare il container e l'ambiente di esecuzione della funzione.

La durata del cold start varia in base al provider e al linguaggi di programmazione utilizzato e, può influire drasticamente sulle prestazioni dell'applicazione.

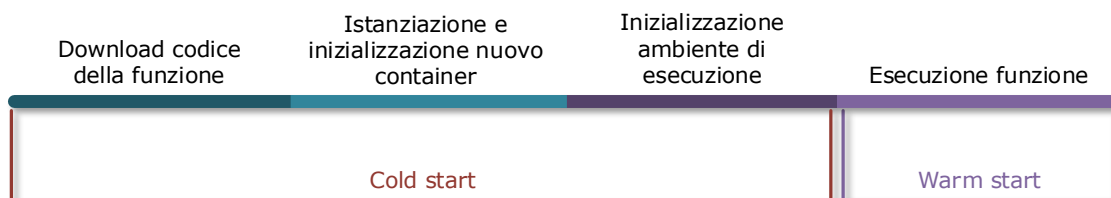


Figura 1.15 – Lifecycle di una funzione.

2. Strumenti impiegati

Lo sviluppo della web application prevede l'implementazione dei framework *Django* per il back end, *Bootstrap* per il front end e la distribuzione su *Amazon Web Services* (AWS). È stata realizzata adottando il linguaggio di programmazione *Python* nella versione 3.6 e il framework *Django* nella versione 2.1.1, conformemente ai prerequisiti richiesti per la distribuzione di un'applicazione *Django* in AWS Elastic Beanstalk, indicati nella guida per sviluppatori inclusa nella documentazione di AWS.

Per la scrittura del codice di *BrainIO* è stato impiegato l'editor di codice sorgente *Visual Studio Code*, sviluppato da Microsoft e distribuito come freeware.

Tramite l'abilitazione del linter *pycodestyle*, uno strumento che analizza il codice sorgente con lo scopo di contrassegnare errori di programmazione, bug, errori stilistici e costrutti sospetti, è stato possibile rendere il codice conforme alla *guida allo stile per il codice Python, PEP 8 (Python Enhancement Proposals)*.

Per poter usufruire dei servizi offerti da AWS all'interno dell'applicazione *Django* è stato impiegato *Boto3*, l'SDK di AWS per Python, che rende disponibili le interfacce per i servizi di AWS.

Per la configurazione dei servizi usufruiti, invece, sono state utilizzate le relative console di gestione e l'interfaccia a riga di comando, installando il package *awsebcli* in un ambiente virtuale Python.

Tramite *MySQL Workbench*, uno strumento visuale di progettazione per database, è stato possibile connettersi all'istanza database MySQL in *Amazon Relational Database Service (RDS)*, necessario vista l'impossibilità di effettuare query sul database direttamente dalla console di gestione di Amazon RDS.

Infine, per quanto concerne le illustrazioni, i grafici e i diagrammi presenti nell'elaborato si è ricorso a *Microsoft Visio Professional 2019* e, a *Draw.io* per fruire della libreria di *AWS Architecture Icons*, se non diversamente specificato nelle note a piè di pagina.

2.1 Amazon Web Services (AWS)

Il colosso statunitense Amazon è stato il primo provider di servizi di cloud computing, introducendo, a metà del 2006, *Amazon Web Services* (AWS), una piattaforma basata sul modello di servizio IaaS, la quale include anche servizi PaaS.

Un numero significativo di grandi aziende e start-up ha, ben presto, usufruito dei servizi di elaborazione forniti dall'infrastruttura di AWS, infatti, già nel 2012 si stima un utilizzo da parte di aziende di circa 200 paesi.

Ad evidenziarne la rapida diffusione della piattaforma, un articolo dell'aprile 2016 di uno dei più importanti quotidiani statunitensi, *The New York Times*, riporta come, al tempo, Amazon abbia registrato uno dei trimestri più redditizi della sua storia, proprio grazie ad AWS:

*“La principale fonte di profitti dell'azienda è Amazon Web Services, l'attività di cloud computing avviata poco più di un decennio fa, che ora è sulla buona strada per generare entrate per oltre 10 miliardi di dollari l'anno.”*⁷

I cinque principi fondamentali della piattaforma sono enunciati nel *canone di architettura AWS*, il quale aiuta gli architetti del cloud a creare infrastrutture sicure, ad alte prestazioni, resilienti ed efficienti per le loro applicazioni:

- *Sicurezza*: riguarda la protezione di informazioni e sistemi. Tra gli argomenti chiave sono inclusi la riservatezza e l'integrità dei dati, l'identificazione delle persone adatte a svolgere determinate attività tramite la gestione dei privilegi, protezione dei sistemi e definizione di controlli per rilevare eventi di sicurezza.

Sicurezza e conformità sono una responsabilità condivisa tra AWS e il cliente. In questo *modello di responsabilità condivisa*, AWS è responsabile della “*sicurezza del cloud*”, ovvero si occupa di proteggere l'infrastruttura globale (componenti hardware e software, reti, ...) sulla quale vengono eseguiti tutti i servizi offerti.

Il cliente, invece, è responsabile della “*sicurezza nel cloud*” e, questa varia in base ai servizi cloud scelti. Ad esempio, il servizio *Amazon Elastic Compute Cloud (EC2)*, è classificato come IaaS e, in quanto tale, richiede tutte le necessarie attività di

⁷ NICK WINGFIELD, 2016. Amazon's Cloud Business Lifts Its Profit to a Record. *The New York Times* [online]. Disponibile su <<https://www.nytimes.com/2016/04/29/technology/amazon-q1-earnings.html>> [Data di accesso: 25/02/2021].

configurazione e gestione della sicurezza. I clienti che distribuiscono un'istanza Amazon EC2 sono responsabili della gestione del sistema operativo guest (inclusi aggiornamenti e patch di sicurezza), di qualsiasi software applicativo o utility installati e della configurazione del firewall fornito da AWS, chiamato *gruppo di sicurezza*.

Per servizi come *Amazon Simple Storage Service (S3)*, invece, i clienti accedono agli endpoint per archiviare e recuperare i dati e, sono responsabili della gestione dei dati, incluse le opzioni di crittografia.

- *Efficienza delle prestazioni*: si concentra su come è possibile eseguire i servizi in modo efficiente e scalabile nel cloud. Tra gli argomenti chiave sono inclusi la selezione delle risorse adeguate, per tipo e dimensione, in base ai requisiti del carico di lavoro, il monitoraggio delle prestazioni e l'elaborazione di decisioni consapevoli per garantire il mantenimento di tale efficienza all'evolversi delle esigenze e delle tecnologie.
- *Affidabilità*: si focalizza sull'assicurare che un carico di lavoro esegua le funzioni pianificate, nei modi e nei tempi previsti. Un carico di lavoro resiliente è in grado di ripristinarsi rapidamente da un errore, per soddisfare le richieste delle aziende e dei clienti.

Al fine di garantire un isolamento del guasto e, quindi, di limitarne il raggio d'azione, AWS offre servizi cloud attraverso una rete di data center su diversi continenti, distribuendo copie ridondanti dei servizi in diverse *regioni AWS*.

Ogni regione è un data center completamente autonomo e, si compone di due o più *zone di disponibilità*, isolate e fisicamente separate all'interno di un'area geografica.

AWS dispone di 77 zone di disponibilità distribuite su 24 regioni in tutto il mondo: Stati Uniti orientali (Ohio, Virginia settentrionale), Stati Uniti occidentali (Oregon, California settentrionale), Canada (Centrale), Sud America (San Paolo), Europa (Francoforte, Londra, Parigi, Irlanda, Milano, Stoccolma), Africa (Città del Capo), Medio Oriente (Bahrein), Cina continentale (Pechino, Ningxia), Asia Pacifico (Singapore, Pechino, Sydney, Tokyo, Seoul, Osaka, Mumbai, Hong Kong).

Inoltre, sono state già annunciate altre 18 zone di disponibilità e 6 ulteriori regioni in Australia, India, Indonesia, Giappone, Spagna e Svizzera.



Figura 2.1 – Regioni AWS. ⁸

La scelta di una regione AWS può essere effettuata tenendo conto di alcuni fattori fondamentali:

- I. *Costi dei servizi*: possono essere diversi per ciascuna regione a causa delle diverse normative di ogni Paese, della valuta e delle spese operative, come, ad esempio, l'elettricità.
- II. *Latenza*: la distanza dell'utilizzatore dalle risorse fisiche potrebbe incidere notevolmente sulle prestazioni dell'applicazione. Pertanto, sarebbe opportuno identificare la posizione dell'utenza, al fine di scegliere la regione AWS più vicina ad essa.
- III. *Sicurezza e conformità*: ciascun Paese ha le proprie regole di sicurezza e conformità, quindi, è necessario verificare i requisiti normativi di una specifica regione. Il *centro conformità AWS* permette, una volta selezionato il Paese di interesse, di visualizzare rapidamente la sua posizione normativa rispetto all'adozione dei servizi cloud.
- IV. *Disponibilità dei servizi*: sebbene la maggior parte dei servizi AWS più diffusi sia disponibile in tutte le regioni, non tutti i servizi lo sono.

⁸ Immagine tratta dalla pagina introduttiva della documentazione di AWS. Disponibile su <https://aws.amazon.com/it/about-aws/global-infrastructure/?pg=WIAWS> [Data di accesso: 26/02/2021].

Nell'ambito dello sviluppo di *BrainIO* è stata scelta la regione di Francoforte, poiché, nonostante Milano fosse la più vicina e, quindi, la migliore in termini di latenza, in questa non sono disponibili alcuni servizi impiegati per l'elaborazione dei video, come *Amazon Transcribe* e *Amazon Comprehend*. Un'altra motivazione che ha indotto ad optare per questa scelta è di carattere economico, infatti, nella regione di Milano non sono disponibili alcune istanze di servizi incluse nel piano gratuito di AWS, come l'istanza di database *db.t2.micro* di *Amazon RDS*.

- *Eccellenza operativa*: si concentra su come migliorare continuamente sistemi, processi e procedure. Tra gli argomenti principali sono inclusi l'automazione di modifiche, la reazione agli eventi e la definizione di standard per la corretta gestione delle operazioni quotidiane.
- *Ottimizzazione dei costi*: è incentrato sul raggiungimento dei risultati aziendali, riducendo al minimo i costi. I servizi cloud di AWS adottano un modello di prezzi calcolati in base all'utilizzo effettivo, in cui è richiesto il pagamento della sola capacità realmente utilizzata.

I servizi di AWS impiegati nello sviluppo della web application sono: *AWS Identity and Access Management (IAM)*, *AWS Elastic Beanstalk*, *Amazon Elastic Compute Cloud (EC2)*, *Amazon Simple Storage Service (S3)*, *Amazon Relational Database Service (RDS)*, *AWS Lambda*, *Amazon Transcribe*, *Amazon Comprehend*.

2.1.1 AWS Identity and Access Management (IAM)

AWS IAM è il servizio che permette di gestire l'accesso alle risorse AWS, mediante la creazione di policy per la concessione di specifiche autorizzazioni ad un'identità IAM (*policy basate su identità*), ossia utenti, gruppi e ruoli IAM, oppure ad una risorsa (*policy basate su risorse*), come ad esempio un bucket *Amazon S3*. L'autorizzazione ad eseguire un'operazione per una determinata risorsa è concessa se la policy basata sull'identità lo consente e, quella basata sulla risorsa, se esiste, non lo proibisce.

Quando si crea un account AWS, viene generato un *utente root*, il quale ha accesso completo a tutti i servizi e le risorse nell'account. È vivamente consigliato non utilizzare tale utente per le attività quotidiane, ma creare più utenti, in modo tale da applicare il *principio del privilegio minimo*, ovvero definire diverse policy per concedere solo le autorizzazioni strettamente necessarie all'espletamento delle proprie funzioni.

Un *utente IAM* rappresenta la persona o il servizio che accede alla risorsa AWS. Viene creato definendo delle credenziali di accesso (nome utente e password) che gli consentono di accedere alla console di gestione AWS e di effettuare richieste programmatiche ai servizi della piattaforma cloud mediante l'interfaccia a riga di comando (CLI), oppure, dall'interno di un'applicazione, tramite l'SDK di AWS per lo specifico linguaggio di programmazione utilizzato. Nonostante ciascun utente goda di proprie credenziali, non sono account AWS distinti, ma risiedono tutti all'interno dell'account root.

Un *gruppo IAM* è definito da un insieme di utenti e permette di semplificare la gestione delle autorizzazioni, specificando delle policy comuni a tutti gli utenti del gruppo.

Un *ruolo IAM* è un'identità simile a un utente, a cui, però, non sono associate delle credenziali di accesso e non identifica in modo univoco una persona o un servizio, ma viene assunto da chiunque ne abbia bisogno. Un utente IAM può assumere un ruolo per ottenere temporaneamente le autorizzazioni necessarie ad una specifica attività.

I ruoli sono utilizzati tipicamente per fornire ad un servizio l'accesso ad un altro servizio AWS. Ad esempio, nella creazione di un'applicazione in un'istanza di Amazon EC2 che effettua richieste ad altri servizi di AWS è preferibile, anziché creare un utente IAM, includendo le sue credenziali nel codice dell'applicazione, definire un ruolo IAM da collegare all'istanza per fornire delle credenziali temporanee. Quando l'applicazione utilizza tali credenziali è in grado di eseguire tutte le operazioni consentite dalle policy associate a quel ruolo.

2.1.2 AWS Elastic Beanstalk

Elastic Beanstalk gestisce automaticamente le funzionalità di distribuzione, provisioning delle risorse, bilanciamento del carico, scalabilità automatica e monitoraggio delle applicazioni, interagendo con altri servizi AWS, tra cui *EC2*, *S3*, *Elastic Load Balancing* e *Auto Scaling*. Permette, quindi, di distribuire e gestire le applicazioni nel cloud senza preoccuparsi dell'infrastruttura che le esegue.

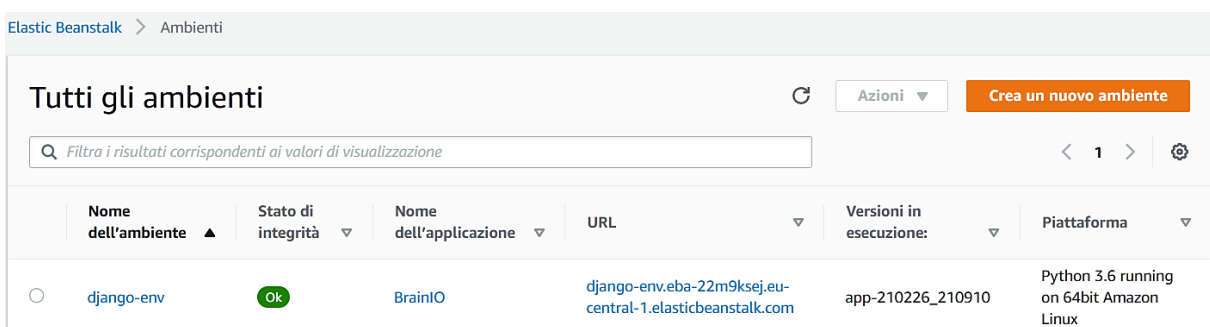
Tale servizio supporta applicazioni e servizi Web sviluppati in Go, Java, .NET, Node.js, PHP, Python, Ruby e Docker.

In Elastic Beanstalk, un'applicazione funge da container per gli ambienti che eseguono le versioni del codice sorgente dell'app stessa, per le configurazioni salvate e i log.

Una volta creata un'applicazione e, caricata una versione di essa, con la definizione di un ambiente, Elastic Beanstalk crea e configura automaticamente le risorse necessarie all'esecuzione del codice. Un ambiente, infatti, è una raccolta di risorse AWS che eseguono una versione dell'applicazione. È possibile creare più ambienti quando si ha la necessità di eseguire più versioni di un'applicazione.

Quest'ultima è accessibile tramite l'URL dell'ambiente in cui è in esecuzione.

Inoltre, è possibile interagire con Elastic Beanstalk utilizzando la console di gestione AWS, l'interfaccia a riga di comando (CLI) oppure *eb*, un'interfaccia a riga di comando di alto livello concepita appositamente per Elastic Beanstalk.



Tutti gli ambienti						
Filtra i risultati corrispondenti ai valori di visualizzazione						
Nome dell'ambiente ▲	Stato di integrità ▼	Nome dell'applicazione ▼	URL ▼	Versioni in esecuzione: ▼	Piattaforma ▼	
○ django-env	OK	BrainIO	django-env.eba-22m9ksej.eu-central-1.elasticbeanstalk.com	app-210226_210910	Python 3.6 running on 64bit Amazon Linux	

Figura 2.2 – La pagina panoramica dell'applicazione BrainIO, nella console di gestione di Elastic Beanstalk, mostra l'elenco degli ambienti associati all'applicazione.⁹

⁹ Schermata della pagina panoramica dell'applicazione nella console di gestione di Elastic Beanstalk.

2.1.3 Amazon Elastic Compute Cloud (EC2)

Amazon EC2 fornisce capacità di elaborazione scalabile nel cloud di AWS, permettendo di avviare il necessario numero di server virtuali, chiamati *istanze*, in funzione di eventuali picchi di richieste, riducendo così la necessità di elaborare previsioni relative al traffico.

Un'*Amazon Machine Image (AMI)* è un modello preconfigurato che contiene una configurazione software (ad esempio un sistema operativo), a partire dal quale è possibile avviare una o più istanze, le quali non sono altro che copie dell'AMI in esecuzione come server virtuale nel cloud.

L'utente sceglie la regione e la zona di disponibilità dove posizionare questo server virtuale, specificando anche, tra quelli disponibili, il *tipo di istanza*, il quale determina l'hardware del computer host utilizzato per tale istanza.

La web application, sviluppata nell'ambito dell'attività virtuale, è stata distribuita utilizzando la piattaforma Python di Elastic Beanstalk in un AMI Amazon Linux a 64 bit.

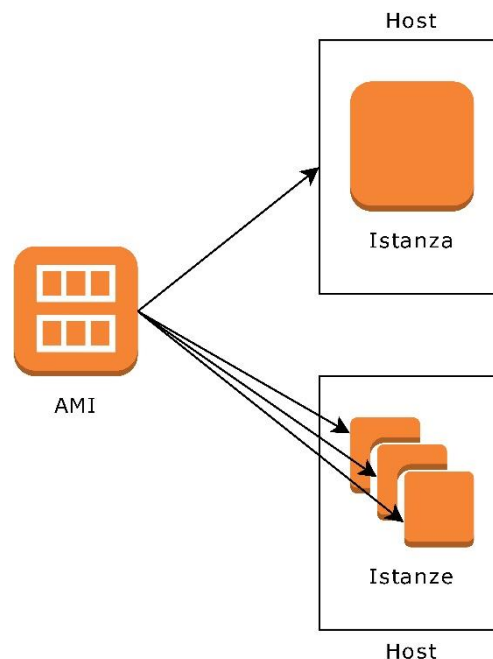


Figura 2.3 – Da un'AMI è possibile avviare una o più istanze.

2.1.4 Amazon Simple Storage Service (S3)

Amazon S3 è un servizio di archiviazione progettato per permettere di archiviare oggetti di grandi dimensioni in container denominati *bucket*. È possibile archiviare una quantità infinita di oggetti in un bucket e, ciascuno di questi può contenere fino a 5 TB di dati.

Il nome di un bucket Amazon S3 è univoco a livello globale e lo spazio dei nomi è condiviso da tutti gli account AWS, in qualsiasi regione.

Naturalmente, per ridurre al minimo i costi o per conformarsi a requisiti normativi è opportuno scegliere una regione geograficamente vicina per ottimizzare la latenza.

Gli *oggetti* archiviati in un bucket sono identificati univocamente da una *chiave* (o *nome di chiave*).

Amazon S3 è caratterizzato da una *struttura flat*, ossia non è presente nessuna gerarchia di bucket secondari o sottocartelle. Tuttavia, è possibile definire una gerarchia logica, applicabile dalla console di gestione di S3 e dagli SDK di AWS per introdurre il concetto di cartella, utilizzando *delimitatori* e *prefissi dei nomi di chiave*.

Ad esempio, supponendo che il bucket *brainiostorage* contenga due oggetti con le seguenti chiavi:

Sessioni/Elaborazioni/trascrizione.json
video.mp4

i prefissi dei nomi di chiavi '*Sessioni*', '*Elaborazioni*' e il delimitatore '/' sono utilizzati per visualizzare una gerarchia di cartelle. Poiché la chiave '*video.mp4*' non è preceduta da nessun prefisso, l'oggetto ad essa associato verrà visualizzato direttamente al livello root del bucket.

È possibile accedere ad un bucket utilizzando l'interfaccia utente della console di gestione, in modo programmatico tramite gli SDK di AWS, oppure tramite URL.

Amazon S3 supporta sia *URL in stile hosting virtuale*:

<https://nome-bucket.s3.Regione.amazonaws.com/chiave-oggetto>

sia *URL in stile percorso*:

<https://s3.Regione.amazonaws.com/nome-bucket/chiave-oggetto>

L'accesso ad un bucket e agli oggetti in esso inclusi viene concesso mediante *liste di controllo accessi (ACL)*, *policy di bucket* o entrambe.

Tra le impostazioni del bucket è possibile configurare un *blocco dell'accesso pubblico*, il quale va a sostituire eventuali ACL o policy, in modo da poter imporre dei limiti all'accesso pubblico per tutti gli oggetti del bucket.

Le *liste di controllo accessi (ACL)* di Amazon S3 sono un meccanismo utilizzato principalmente prima dell'introduzione del servizio IAM. Tuttavia, sono ancora necessarie in alcuni scenari di controllo degli accessi, come, ad esempio, nel momento in cui il proprietario del bucket debba concedere delle autorizzazioni ad oggetti che non sono di sua proprietà. In tal caso, il proprietario dell'oggetto deve prima concedere le opportune autorizzazioni al proprietario del bucket. Ciò avviene mediante la definizione di ACL.

Le *policy del bucket* permettono di garantire un controllo degli accessi centralizzato a bucket e ad oggetti in base a varie condizioni, tra le quali operazioni di Amazon S3, richiedenti del servizio, risorse e parametri della richiesta, come l'indirizzo IP.

Le autorizzazioni concesse tramite policy del bucket si applicano a tutti gli oggetti appartenenti al proprietario del bucket oppure ad un sottoinsieme di essi, a differenza delle ACL, le quali permettono di concedere autorizzazioni solamente a singoli oggetti.

Infine, è possibile condividere le risorse con un insieme limitato di persone definendo un gruppo IAM con le relative policy, per poi andare a creare i vari utenti IAM da inserire in tale gruppo.

È consigliato, come regola generale, impiegare per il controllo degli accessi, policy del bucket o policy IAM.

2.1.5 Amazon Relational Database Service (RDS)

Amazon RDS è un servizio che semplifica la configurazione, l'utilizzo e il dimensionamento di un database relazionale nel cloud AWS.

Offre una capacità ridimensionabile ad un costo conveniente, automatizzando la gestione di backup, l'applicazione di patch di aggiornamento, la rilevazione dei guasti e il relativo ripristino. È possibile configurare backup automatici in base alle proprie esigenze oppure creare manualmente snapshot di backup, utili anche per ripristinare una versione precedente del database.

Garantisce, inoltre, un'elevata disponibilità permettendo la definizione di un'istanza principale ed una secondaria sincrona, in un'altra zona di disponibilità, sulla quale mantenere una replica dei dati, utile per eseguire il failover in caso di problemi.

Un'*istanza database* rappresenta l'elemento base di Amazon RDS ed è un ambiente di database isolato in esecuzione nel cloud. Questa può anche contenere più database creati dall'utente.

Ciascuna istanza dispone di un identificatore univoco, con il quale RDS definisce automaticamente un endpoint per accedervi tramite interfaccia a riga di comando oppure connettendosi da un'applicazione di terze parti, come MySQL Workbench.

Ad esempio, un endpoint potrebbe essere:

`db1.123456789012.us-east-1.rds.amazonaws.com`

dove '*db1*' è l'identificatore dell'istanza DB, mentre '*123456789012*' è l'identificatore fisso della specifica regione selezionata.

Un'istanza esegue un *motore di database*, tra i quali, sono attualmente supportati MySQL, MariaDB, PostgreSQL, Oracle e Microsoft SQL Server.

2.1.6 AWS Lambda

AWS Lambda è un servizio di calcolo che consente di eseguire codice senza gestire i server o effettuarne il provisioning, ovvero permette un'elaborazione *serverless*.

Lambda è una piattaforma FaaS nella quale è possibile creare una *funzione*, in uno dei linguaggi di programmazione supportati (Python, Ruby, Java, Go, C#). Il codice della funzione può essere distribuito tramite un *pacchetto di distribuzione*, che può essere un'immagine di container oppure un archivio di file .zip e, AWS Lambda crea automaticamente un container per la sua esecuzione, ogni volta che questa viene invocata.

Le funzioni Lambda sono *event-driven*, ovvero vengono invocate in risposta ad un evento *trigger*, il quale può verificarsi in altri servizi di AWS, come ad esempio la creazione di un oggetto in un bucket Amazon S3, in modo da elaborare i dati immediatamente dopo il loro caricamento.

Oltre a definire il metodo *handler*, che viene eseguito quando la funzione viene invocata, nella console di gestione di AWS Lambda è possibile configurare diversi parametri della funzione, come la quantità di memoria da riservare per la sua esecuzione (valore compreso tra 128 MB e 10240 MB), un timeout massimo di 15 minuti e il ruolo IAM da associare.

AWS Lambda monitora automaticamente le funzioni Lambda per conto dell'utente e fornisce i parametri della funzione ad un altro servizio, *Amazon CloudWatch*, il quale genera dei *gruppi di log* con un flusso di log per ogni istanza della funzione.

È possibile generare i log dal codice della funzione utilizzando il metodo *print()*, oppure un qualsiasi altro metodo che permetta di scrivere su *stdout* o *stderr*.

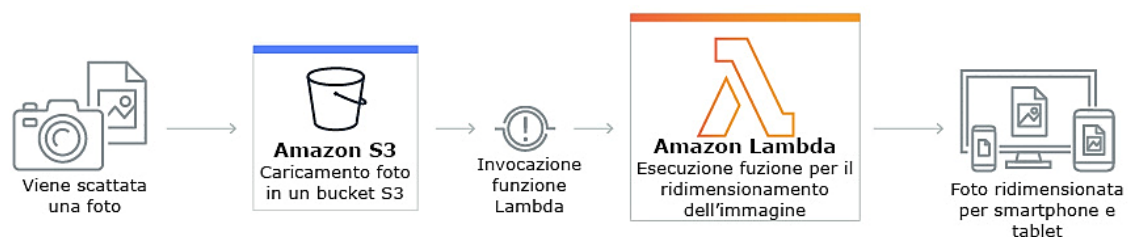


Figura 2.4 – Esempio di invocazione di una funzione Lambda, configurando come trigger il caricamento di un oggetto in un bucket Amazon S3. ¹⁰

¹⁰ Immagine tratta dalla pagina introduttiva del servizio AWS Lambda.
Disponibile su <<https://aws.amazon.com/it/lambda/>> [Data di accesso: 01/03/2021].

2.1.7 Amazon Transcribe

Amazon Transcribe utilizza un processo di deep learning, chiamato riconoscimento vocale automatico (ASR), per riconoscere i dati vocali in un file audio o video e trascriverli in testo.

Supporta file in formato FLAC, MP3, MP4, Ogg, WebMAMR e WAV, della dimensione massima di 2 GB e lunghezza compresa tra 500 millisecondi e 4 ore. Tuttavia, per ottenere risultati ottimali è consigliabile utilizzare un formato senza perdita di dati, come FLAC oppure WAV con codifica PCM a 16 bit e, una frequenza di campionamento di 8000 Hz per una bassa fedeltà audio o di 16000 Hz per un audio ad alta fedeltà.

La trascrizione può essere avviata utilizzando l'SDK di AWS oppure l'interfaccia a riga di comando tramite l'operazione *StartTranscriptionJob*, la quale avvia un processo batch per trascrivere la sintesi vocale di un file audio o video, in testo.

Quest'ultima permette anche di specificare diversi parametri, tra i quali la posizione del file multimediale all'interno di un bucket S3 (*MediaFileUri*), il nome del file JSON che conterrà l'output della trascrizione e, per una maggiore precisione nell'identificazione della lingua, una matrice delle lingue parlate nel file in input. Se non si specifica un set di lingue, vengono elencati, nel file risultante, i codici delle cinque lingue che hanno ottenuto un punteggio di affidabilità più alto tra quelle identificate e, trascrive l'audio nella lingua con il punteggio di affidabilità più elevato.

Amazon Transcribe aggiunge automaticamente la punteggiatura e una formattazione adeguata, in modo da ottenere un testo chiaro e facilmente comprensibile, offrendo la possibilità di fornire un *vocabolario personalizzato*, in cui includere un elenco di parole specifiche che il servizio deve riconoscere nell'input audio.

2.1.8 Amazon Comprehend

Amazon Comprehend è un servizio di elaborazione del linguaggio naturale che adotta l'apprendimento automatico per trovare informazioni in qualsiasi tipo di file di testo in formato UTF-8.

Permette di lavorare su uno o più documenti contemporaneamente, al fine di analizzarne il contenuto ed estrapolarne informazioni del tipo:

- *Entità*: riferimento testuale al nome di oggetti del mondo reale, come persone, luoghi e articoli commerciali oppure riferimenti ad altre informazioni, come date e quantità.
- *Frase chiave*: è una stringa contenente una frase nominale che descrive un qualcosa di particolare.

Generalmente è costituita da un nome e da aggettivi qualificativi che la contraddistinguono. Ad esempio, “*giornata*” è un sostantivo, mentre “*una bellissima giornata*” è una frase nominale.

Ad ogni frase chiave viene assegnato un punteggio, il quale indica il livello di affidabilità che la stringa sia una frase nominale.

È possibile utilizzare tale punteggio per determinare se il rilevamento è sufficientemente affidabile per l'applicazione.

- *Informazioni di identificazione personale*: riferimenti testuali a dati personali che potrebbero essere utilizzati per identificare un individuo, come ad esempio un numero di conto bancario, un numero telefonico o un indirizzo.
- *Lingua*: identifica la lingua dominante in un documento.
- *Sintassi*: analizza il documento al fine di restituire la funzione sintattica di ciascuna parola che lo compone. Può identificare nomi, verbi, aggettivi e così via.
- *Analisi delle emozioni*: determina il “sentimento” di un documento, il quale può essere distinto in quattro categorie: positivo, negativo, neutro oppure misto.

Il risultato di tale elaborazione riporta lo stato emotivo più probabile del testo analizzato ed i punteggi nelle relative categorie.

Nell'esempio seguente, si ha un probabilità del 95% che il sentimento prevalente del documento sia positivo, meno dell'1% che sia negativo, più dell'1% che sia neutrale e circa del 3% che sia misto.

```
{
  "SentimentScore": {
    "Mixed": 0.030585512690246105,
    "Positive": 0.94992071056365967,
    "Neutral": 0.0141543131828308,
    "Negative": 0.00893945890665054
  },
  "Sentiment": "POSITIVE",
  "LanguageCode": "it"
}
```

2.2 Django Web Framework

Django è un web framework di alto livello con licenza open source, scritto nel linguaggio di programmazione Python, che consente un rapido sviluppo di web application sicure e manutenibili.

Il framework adotta la filosofia “*batterie incluse*” tipica di Python, che gli permette di fornire tutto il necessario per sviluppare un’applicazione completa di tutto, implementando dei processi comuni, ma complessi, inclusi nei package *contrib*, tra i quali i più utilizzati sono *admin*, *auth*, *messages*, *staticfiles*.

Da Python eredita anche l’elevata portabilità, la quale consente ad un’applicazione sviluppata con tale framework di funzionare su tutte le piattaforme server.

Django è stato sviluppato con un’elevata attenzione alla sicurezza, fornendo un ottimo supporto agli sviluppatori nell’evitare errori di sicurezza molto comuni. Ad esempio, permette di evitare l’errore di memorizzare direttamente la password, piuttosto che l’hash di questa, utilizzando automaticamente degli algoritmi di crittografia (PBKDF2 di default) per memorizzarla in modo sicuro nel database.

Inoltre, offre di default una protezione contro molte vulnerabilità, come SQL injection, cross-site scripting, cross-site request forgery e clickjacking.

Un progetto Django è definito da un insieme di *applicazioni* e impostazioni di configurazione. Ciascuna applicazione si occupa di uno specifico compito.

L’architettura del framework è basata sul design pattern *Model-View-Controller (MVC)*, il quale porta a suddividere l’applicazione software in tre componenti interconnessi, separando la logica di presentazione dei dati dalla logica di business:

- *Model*: fornisce l’interfaccia con il database contenente i dati dell’applicazione.
- *View*: decide quali informazioni passare all’utente e raccoglie informazioni da esso, gestendo l’interazione fra quest’ultimo e l’infrastruttura sottostante.
- *Controller*: gestisce la logica di business dell’applicazione, ricevendo le richieste dell’utente attraverso la View e rispondendo dopo aver eseguito operazioni che potrebbero interessare il Model.

In realtà, Django utilizza una terminologia leggermente diversa nella sua implementazione del pattern (mostrato in *Figura 2.6*), ovvero *Model-Template-View*

(MTV), nella quale è il framework stesso ad agire da Controller, inviando una richiesta alla View appropriata, in base alla configurazione degli URL.

In Django:

- *Model*: fornisce una mappatura tra un oggetto e il database sottostante (*Object-relational mapping*, *ORM*). In tal modo lo sviluppatore non ha bisogno di conoscere la struttura del database e non necessita di query SQL per manipolare e recuperare i dati.

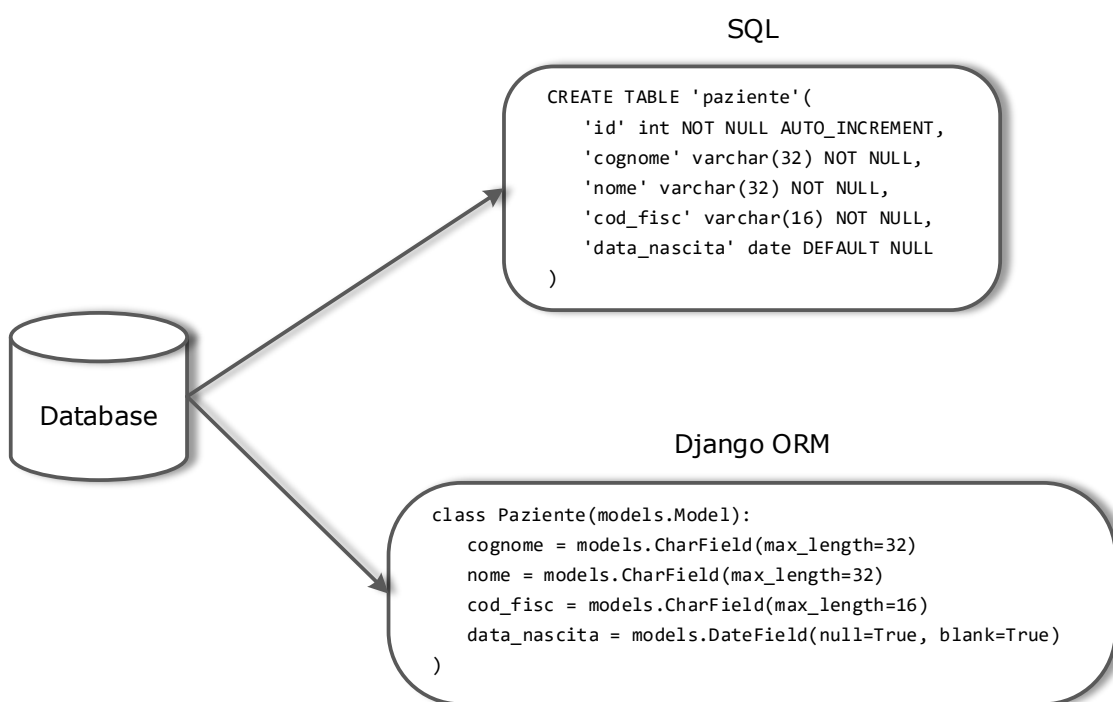


Figura 2.5 – Un ORM consente una semplice manipolazione dei dati, senza dover scrivere query SQL complesse.

Ciascun model è una classe che rappresenta una tabella del database e, ogni suo attributo corrisponde ad un campo della tabella. Vengono definiti nel file *models.py* di un'applicazione Django e sono implementati come sottoclassi di *django.db.models.Model*.

Il nome del model coinciderà con il nome della tabella nel database, al quale Django aggiungerà automaticamente un prefisso corrispondente al nome dell'applicazione Django alla quale appartiene.

Il framework Django supporta ufficialmente cinque tipi di database: PostgreSQL, MariaDB, MySQL, Oracle e SQLite. Tuttavia, è possibile ricorrere a soluzioni di terze parti per includere database non supportati ufficialmente.

- *Template*: fornisce la logica di presentazione ed è l'interfaccia tra l'utente e l'applicazione. È un file di testo progettato, appunto, per separare i dati di un'applicazione, dal modo in cui questi vengono presentati.

Generalmente scritto in HTML, nonostante possa essere effettuato anche per il rendering di altri formati testuali, permette di generare pagine web dinamiche mediante l'inserimento di appositi *tag* oppure di valori passati dalla View in un dizionario, denominato *context*.

Infatti, in un file HTML non è possibile scrivere direttamente codice Python, poiché questo può essere interpretato solamente dall'interprete Python.

Django incoraggia lo sviluppo di codice manutenibile e riutilizzabile, in linea con il principio di progettazione *DRY (Don't Repeat Yourself)*, offrendo la possibilità di sfruttare l'*ereditarietà tra Template*, in modo da evitare inutili duplicazioni del codice ed eventuali bug.

- *View*: è una funzione Python che riceve le richieste HTTP e restituisce le relative risposte. Ogni View svolge una funzione specifica ed è associata ad un determinato Template.

Una View può creare dinamicamente una pagina HTML utilizzando un Template, popolandola con i dati di un Model.

La navigazione in un sito web Django è analoga a qualsiasi altro sito web, ovvero le pagine e gli altri contenuti sono accessibili tramite un *URL (Uniform Resource Locator)*. Una volta che Django riceve l'URL richiesto, deve decidere quale View si occuperà della richiesta HTTP. Il programmatore decide quale View servire ad uno specifico URL creando una *URLconf (URL configuration)* in un file Python, chiamato *urls.py*.

Quando Django trova un URL in tale file che corrisponde a quello richiesto, chiama la View associata a tale URL. Quest'ultima, effettua il rendering del contenuto in un Template, il quale si occupa di formattare adeguatamente la risposta ed inviarla al browser per la visualizzazione.

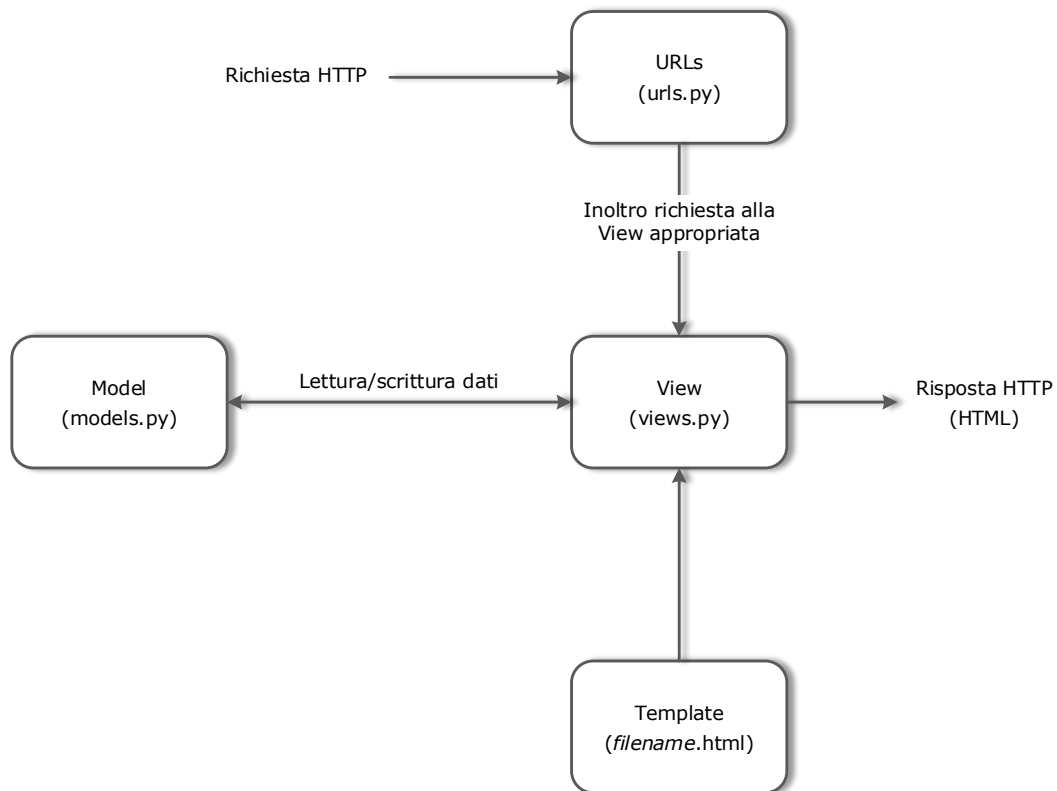


Figura 2.6 – Rappresentazione del modello Model-Template-View (MTV), implementazione di Django del design pattern MVC.

2.3 Bootstrap Framework

Bootstrap, ideato originariamente da un gruppo di sviluppatori di Twitter come strumento per uniformare i vari web design, è diventato ben presto, dopo essere stato rilasciato sulla piattaforma open source GitHub, il front-end framework più popolare per la realizzazione di web application caratterizzate da un *responsive* oppure da un *mobile-first web design*.

Tale framework offre diversi modelli di progettazione basati su HTML e CSS, oltre a molteplici plug-in opzionali JavaScript, basati sulla libreria jQuery, i quali possono essere facilmente integrati in un Template Django.

Tali modelli consentono di ridurre notevolmente le necessarie configurazioni CSS, fornendo diversi elementi sia per la tipografia, che per le svariate componenti dell'interfaccia, come pulsanti, elementi di navigazione, menu a tendina, popover, card, elementi per la paginazione, alert per gli errori e molti altri.

La realizzazione di un responsive web design è semplificata dall'impiego del *Grid System* di Bootstrap, il quale consente di stabilire esattamente le distanze e il posizionamento degli elementi nella pagina, suddividendo il layout in 12 colonne e scalando automaticamente all'aumentare delle dimensioni del dispositivo o semplicemente della finestra del browser.

Il layout della pagina sarà definito da una serie di righe, utilizzate per creare gruppi orizzontali di colonne e contenute all'interno di *container*, i quali costituiscono l'elemento fondamentale del Grid System e, garantiscono un corretto allineamento e riempimento del contenuto.

Bootstrap permette di definire tre tipi di container:

- *.container*: imposta una larghezza fissa per ciascun breakpoint.
- *.container-fluid*: adatta la larghezza occupando l'intera area a sua disposizione.
- *.container-{breakpoint}*: imposta la larghezza alla dimensione massima del breakpoint specificato.

Il framework include di default sei *breakpoint*, progettati per contenere comodamente i container e le cui larghezze sono multiple di 12.

Questi sono anche rappresentativi di un sottoinsieme di dimensioni comuni dei dispositivi:

<i>Breakpoint</i>	<i>Infisso classe</i>	<i>Dimensioni</i>
Extra small		< 576px
Small	sm	≥ 576px
Medium	md	≥ 768px
Large	lg	≥ 992px
Extra large	xl	≥ 1200px
Extra extra large	xxl	≥ 1400px

I breakpoint si utilizzano anche per impostare la larghezza di una colonna, la quale può anche essere diversa per ciascuna classe di colonna specificata:

	<i>xs</i> < 576px	<i>sm</i> ≥ 576px	<i>md</i> ≥ 768px	<i>lg</i> ≥ 992px	<i>xl</i> ≥ 1200px	<i>xxl</i> ≥ 1400px
<i>Larghezza massima container</i>	(auto)	540px	720px	960px	1140px	1320px
<i>Prefisso classe</i>	.col-	.col-sm-	.col-md-	.col-lg-	.col-xl-	.col-xxl-

3. BrainIO

La web application *BrainIO*, sviluppata nell'ambito dell'attività progettuale, è una piattaforma software finalizzata all'impiego in ambito psicologico, come strumento di analisi dell'andamento terapeutico dei pazienti.

L'applicazione crea un dataset con i risultati dell'elaborazione delle registrazioni video delle sessioni del paziente, al fine di fornire al professionista sanitario una rappresentazione grafica, sintetica oppure dettagliata, che consenta di cogliere lo stato emotivo del paziente per valutarne i miglioramenti o, eventualmente, i peggioramenti dal punto di vista terapeutico.

Progettata con l'obiettivo di ottenere un'elevata modularità, scalabilità e manutenibilità, in modo da poter essere facilmente ampliata in futuro, BrainIO, può essere utilizzata su qualsiasi dispositivo, dagli smartphone ai tablet, passando per i computer desktop.

Infatti, il responsive web design, realizzato mediante l'implementazione del framework Bootstrap, permette all'applicazione di scalare automaticamente la dimensione dei componenti dell'interfaccia utente quando si passa da un dispositivo all'altro, oppure quando si ridimensiona la finestra del browser.

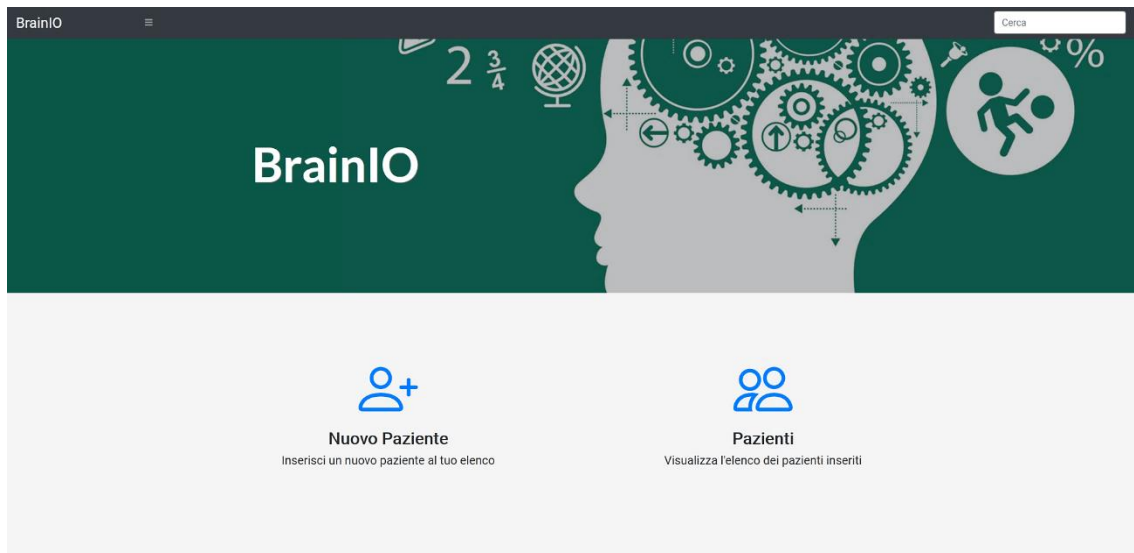


Figura 3.1 – Visualizzazione home page di BrainIO da un computer desktop.

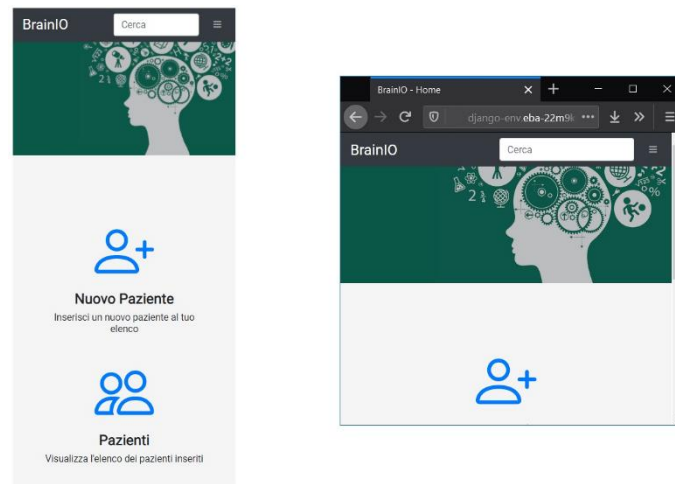


Figura 3.2 – Visualizzazione home page di BrainIO da uno smartphone (a sinistra) e da una finestra di un browser di grandezza ridotta (a destra).

Tutto ciò conferisce una notevole portabilità all'applicazione, la quale è stata ideata con la prerogativa fondamentale della semplicità di utilizzo.

L'interfaccia grafica semplice ed intuitiva favorisce un utilizzo soddisfacente da parte di qualsiasi utente (psicologo).

Non è richiesta, infatti, nessuna competenza informatica, ma è sufficiente una basilare dimestichezza con uno smartphone o un qualunque dispositivo elettronico di uso comune.

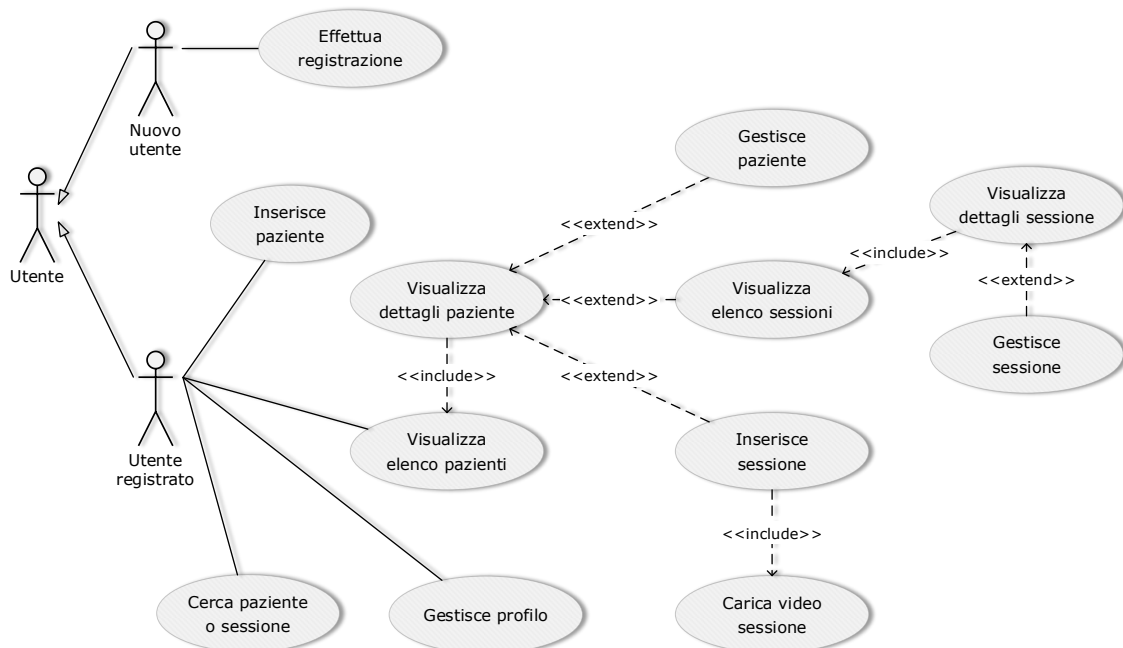


Figura 3.3 – Use Case Diagram di BrainIO.

3.1 Interfaccia sistema/utente

La prima pagina mostrata all'utente che si accinge ad utilizzare la web application è quella del *login*, la quale gli consente di accedere alla sua area personale.

Se non si è in possesso di credenziali di accesso, sarà necessario registrarsi mediante un apposito form, in cui oltre allo username, al nome e al cognome, verrà richiesto un indirizzo di posta elettronica ed una password.

L'account resterà disabilitato finché l'utente non confermerà il proprio indirizzo email, accedendo al proprio account di posta elettronica e cliccando sul link ricevuto. Al fine di garantire una maggiore protezione dei dati personali dell'utilizzatore, la password da inserire in fase di registrazione deve rispettare dei criteri di sicurezza, ovvero non deve essere troppo simile alle altre informazioni inserite, deve contenere almeno otto caratteri, non può essere una password comunemente utilizzata e non può essere interamente numerica.

Se quest'ultima dovesse essere dimenticata, potrebbe essere recuperata facilmente inserendo l'indirizzo email con il quale l'utente si è registrato e seguendo la procedura indicata nel link inviato all'indirizzo di posta elettronica.

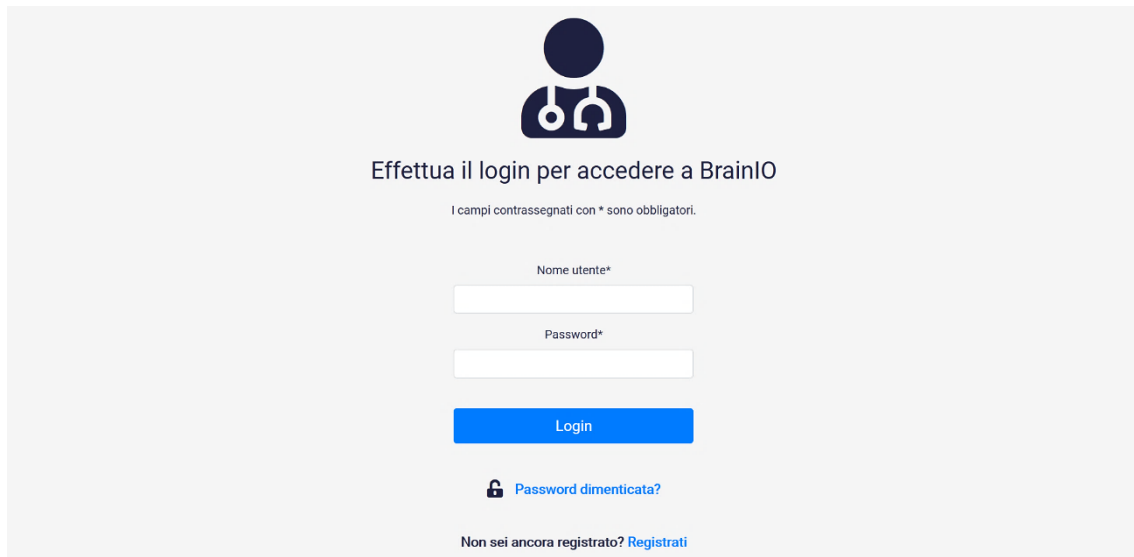


Figura 3.4 – Pagina di Login di BrainIO.

Una volta effettuato l'accesso si verrà reindirizzati alla *home page* dell'applicazione, nella quale risaltano subito all'occhio due pulsanti, uno per inserire un nuovo paziente e l'altro per visualizzare l'elenco dei pazienti inseriti.

Il processo di inserimento di un nuovo paziente necessita della compilazione di un form, nel quale vengono richiesti i dati personali di quest'ultimo, ovvero nome, cognome e codice fiscale.

Premendo sul pulsante di conferma, la piattaforma effettua immediatamente un controllo sul codice fiscale inserito, andando a verificare se i primi sei caratteri, identificativi del nome e del cognome del paziente, coincidono con quelli calcolati a partire dai dati inseriti nel form.

In caso di discordanza verrà visualizzato un messaggio di errore, con la conseguente richiesta di una reiterazione dell'immissione dei dati.

Se, invece, la verifica dovesse avere esito positivo e, non dovessero esserci già altri pazienti con gli stessi dati, allora, a partire dal codice fiscale, verrebbero calcolati automaticamente le altre informazioni anagrafiche e si verrebbe reindirizzati alla pagina riepilogativa.

L'*elenco dei pazienti* inseriti, accessibile premendo l'altro pulsante presente sulla home dell'applicazione, consente allo psicologo di avere una visione chiara e sintetica di alcuni dettagli di ciascuno di essi, visualizzando informazioni come l'ultima sessione effettuata oppure il numero delle sessioni archiviate. Tale lista può essere ordinata in modo ascendente o discendente per nome, cognome, data di inserimento o ultima sessione.

Le *card* mostrate in tale elenco, rappresentative di ciascun paziente, sono composte da un header, contenente il nome e il cognome del soggetto, al fianco di un'icona caratterizzata da uno sfondo azzurro se il paziente è di sesso maschile, viola diversamente, una parte centrale con le informazioni sopracitate e, nella parte inferiore, un footer con la data di inserimento del paziente.

Premendo sull'header della card, il quale assume una colorazione più scura al passaggio del mouse, è possibile accedere alla pagina riservata ai dettagli del paziente.

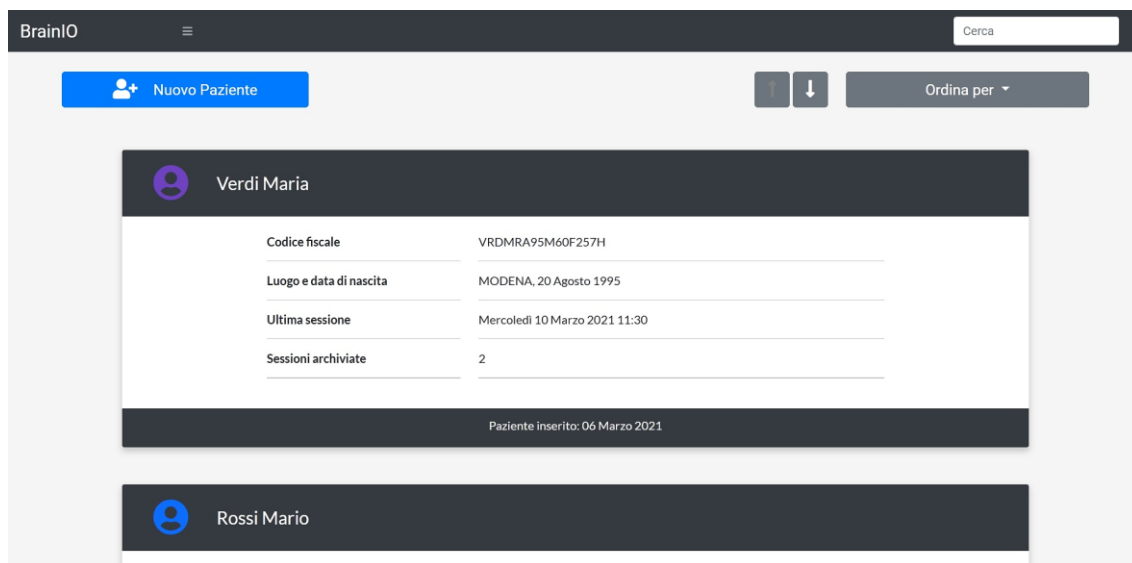


Figura 3.5 – Elenco dei pazienti inseriti.

La *Navigation Bar* nella parte alta consente di effettuare una ricerca del testo inserito tra le informazioni dei pazienti e delle sessioni e, si compone, inoltre, di un pulsante che consente di aprire o chiudere un *Side Navigation Menu* sulla sinistra.

All'apertura del menu laterale, durante la visualizzazione dell'applicazione da computer desktop, il contenuto della pagina si ridimensiona automaticamente, andando ad occupare solo la parte alla destra del menu. Mentre, nei dispositivi più piccoli oppure ridimensionando la finestra del browser, questo si sovrappone oscurando parte del contenuto.

Da tale menu è possibile effettuare il logout, accedere alla lista dei pazienti inseriti o al profilo dello psicologo, in cui, oltre alla modifica dei dati dell'utente, è possibile modificare la password oppure cancellare definitivamente l'account.

Contestualmente all'eliminazione dell'account, vengono automaticamente rimossi tutti i pazienti e le relative sessioni, cancellando sia le opportune righe dal database, che le cartelle nel bucket di Amazon S3, contenenti i risultati dell'elaborazione dei video delle sessioni.

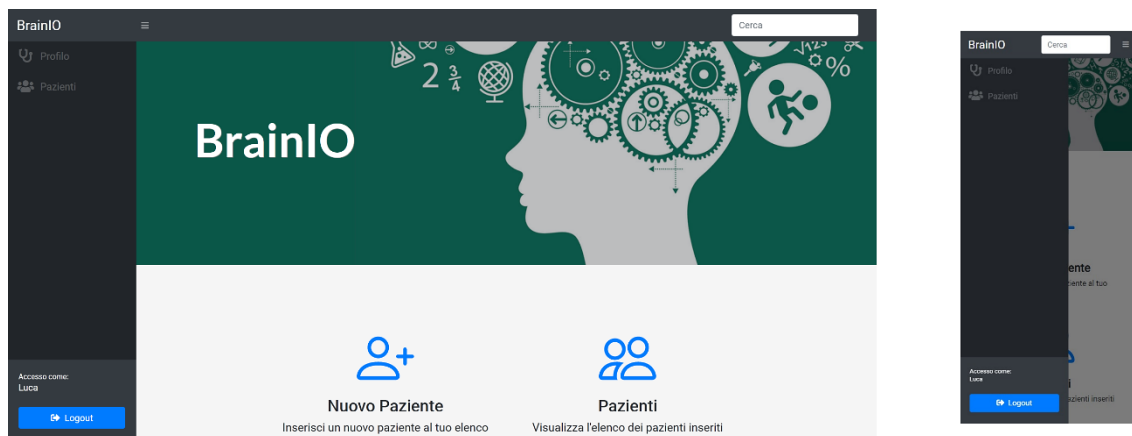


Figura 3.6 – Visualizzazione da browser (sulla sinistra) e da smartphone (sulla destra) della home page con il Side Navigation Menu aperto.

La pagina di visualizzazione dei dettagli del paziente permette la modifica o la cancellazione del paziente stesso e, presenta un’*interfaccia a tab* composta da tre pannelli:

- *Grafici*: riporta i grafici riepilogativi dell’analisi dei sentimenti e delle voci relativamente alle ultime N sessioni.

La scelta del numero delle sessioni delle quali si desidera visualizzare i grafici può essere effettuata dal menu a tendina posto nell’header della card. Tale numero viene popolato con valori che vanno da zero al numero di sessioni con elaborazioni completate, con un passo di avanzamento di 5.

Le elaborazioni sul video di una sessione vengono completate solamente dopo aver selezionato, tra quelle percepite, la voce del paziente, nell’apposita pagina di modifica della sessione.

Il grafico relativo all’analisi dei sentimenti mostra delle barre verticali, una per ogni sessione e, ciascuna di esse può essere suddivisa in quattro parti di colore diverso e di altezza proporzionale alla probabilità che il “*sentimento*” del documento, trascritto mediante Amazon Transcribe, sia di una delle quattro categorie distinte dall’*analisi delle emozioni* di Amazon Comprehend, come descritto precedentemente.

Il secondo grafico rappresenta tre barre verticali per ciascuna sessione, le quali mostrano la quantità di tempo in cui ha parlato il paziente, lo psicologo oppure nessuno dei due.

Il rombo nero su ciascun gruppo di barre specifica la durata totale del video della sessione.

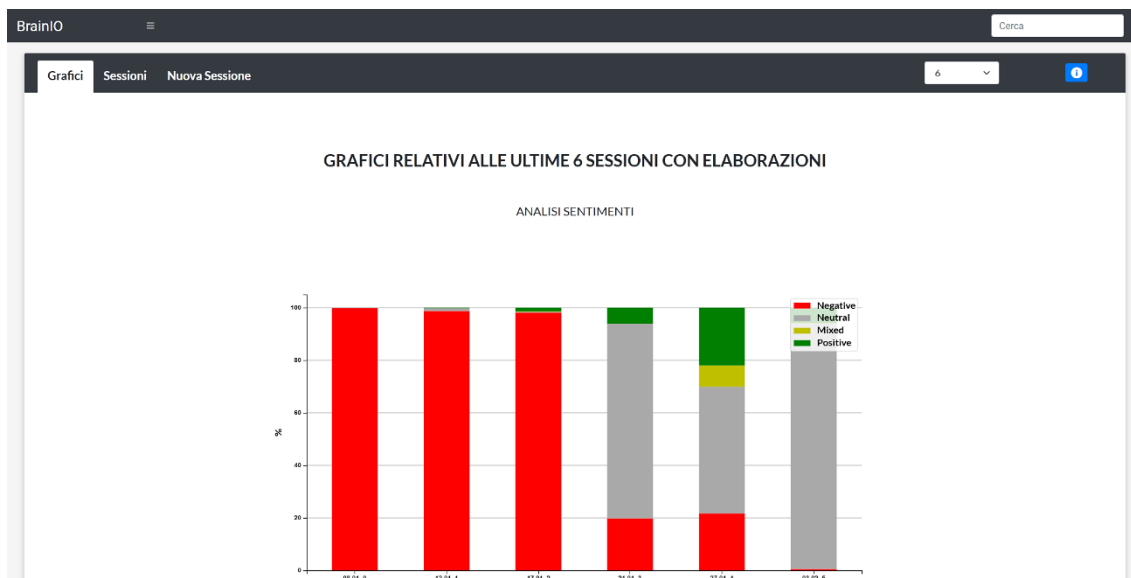


Figura 3.7 – Grafico riepilogativo dell’analisi dei sentimenti delle ultime sei sessioni inserite.



Figura 3.8 – Grafico riepilogativo dell’analisi delle voci delle ultime sei sessioni inserite.

Premendo sul pulsante azzurro nell’header, in alto a destra (come è possibile vedere in *Figura 3.7*), si aprirà un popover contenente una breve descrizione dei grafici raffigurati. Tale pulsante sarà automaticamente nascosto quando si visualizza uno degli altri due pannelli.

- *Sessioni*: viene visualizzato l’elenco delle sessioni archiviate del paziente, mostrando per ciascuna di esse i dettagli relativi alle elaborazioni effettuate, le note e gli hashtag

inseriti, segnalando con un messaggio in rosso se la voce del paziente non è stata ancora selezionata e, consentendo l'eliminazione di una o più sessioni contemporaneamente.

Premendo su una delle sessioni è possibile aprire la pagina di dettaglio di questa, alla quale è possibile accedervi solamente se tutte le funzioni Lambda e, quindi tutte le elaborazioni sul video della sessione, sono terminate.

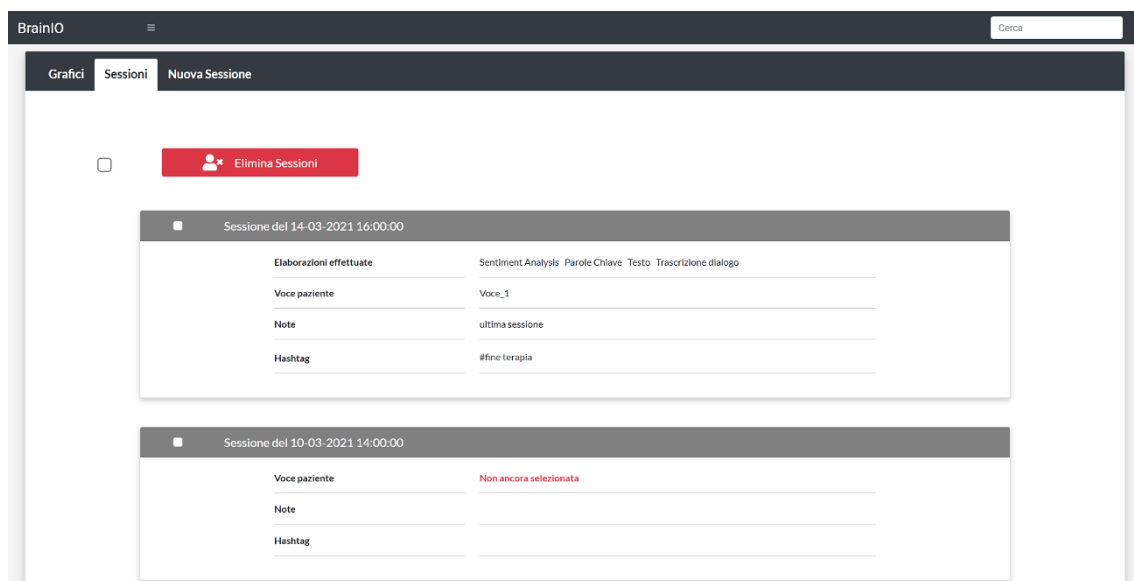


Figura 3.9 – Elenco sessioni archiviate di uno specifico paziente.

- *Nuova sessione*: permette di inserire una nuova sessione, caricando un video e definendo data e ora ed eventuali note e hashtag.

La pagina dedicata ai dettagli di una sessione, oltre a mostrare informazioni su di essa, consente di gestirla, offrendo la possibilità di modificarla oppure eliminarla completamente.

Si compone, inoltre, di un'interfaccia a tab con quattro pannelli.

Nei primi due sono rappresentate graficamente le stesse elaborazioni presenti nella pagina precedentemente descritta, ma in maniera più dettagliata.

Gli altri due, invece, riportano la trascrizione del dialogo tra il paziente e lo psicologo e l'elenco delle frasi chiave, individuate con l'apposita funzione di Amazon Comprehend.

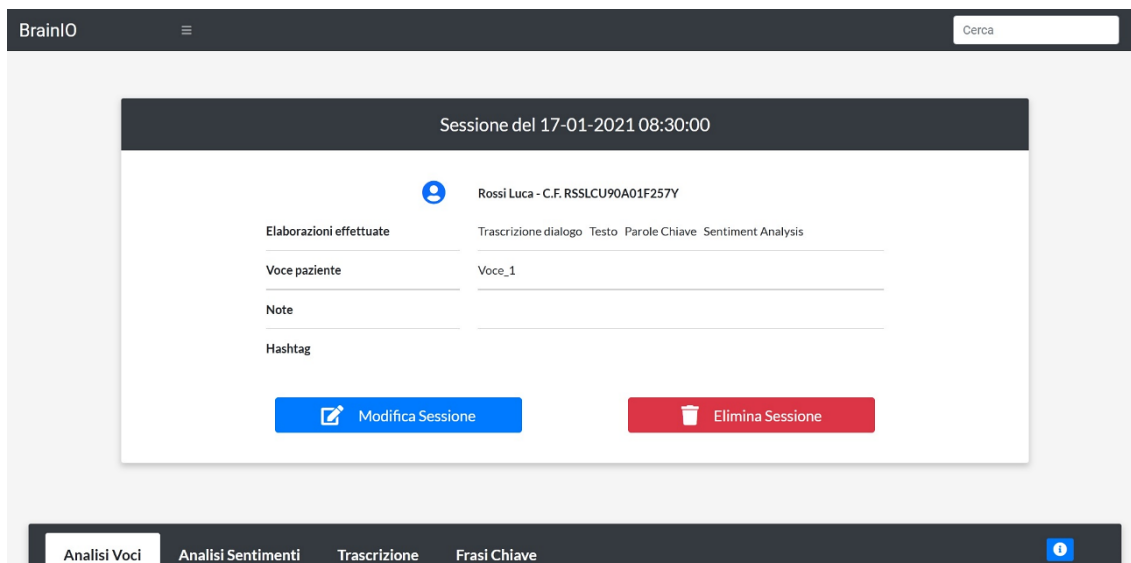


Figura 3.10 – Pagina per la visualizzazione dei dettagli di una sessione.

Nel modificare una sessione, è possibile cambiare data e ora, scegliere un altro paziente dall'apposito menu a tendina, fornire nuove note e hashtag, oppure selezionare, tra quelle identificate nel video, la voce del paziente.

Premendo sull'elemento richiudibile (*collapse*) verranno visualizzate due voci, tra le quali è possibile scegliere quella appartenente al paziente. Inoltre, cliccando su una di queste, sarà mostrato il primo segmento della voce selezionata nella trascrizione del dialogo. Se questo non è sufficiente, nel caso in cui fossero presenti altri segmenti, sarà possibile visualizzarli premendo su *“Visualizza altro”*.

In assenza di segmenti per la voce selezionata, verrà mostrato il messaggio *“Nessun intervento nel dialogo”*.

In seguito alla modifica di una sessione verranno spostati tutti i relativi file nel bucket di Amazon S3 e aggiornate le righe interessate nella tabella *manager_keyvalue* del database.

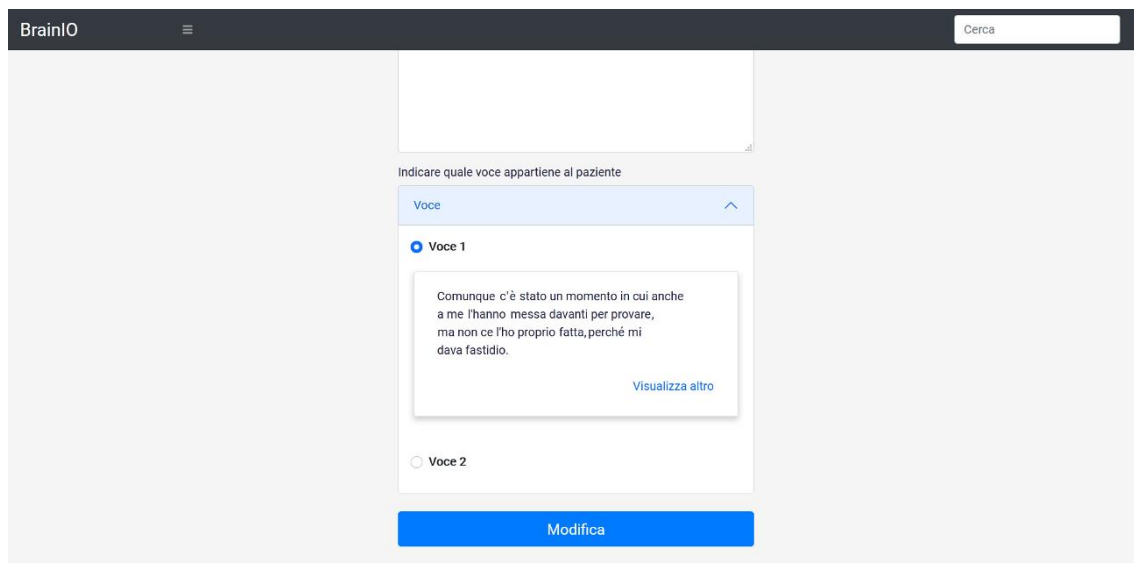


Figura 3.11 – Scelta della voce appartenente al paziente, nella pagina per la modifica di una sessione.

3.2 Struttura progetto Django

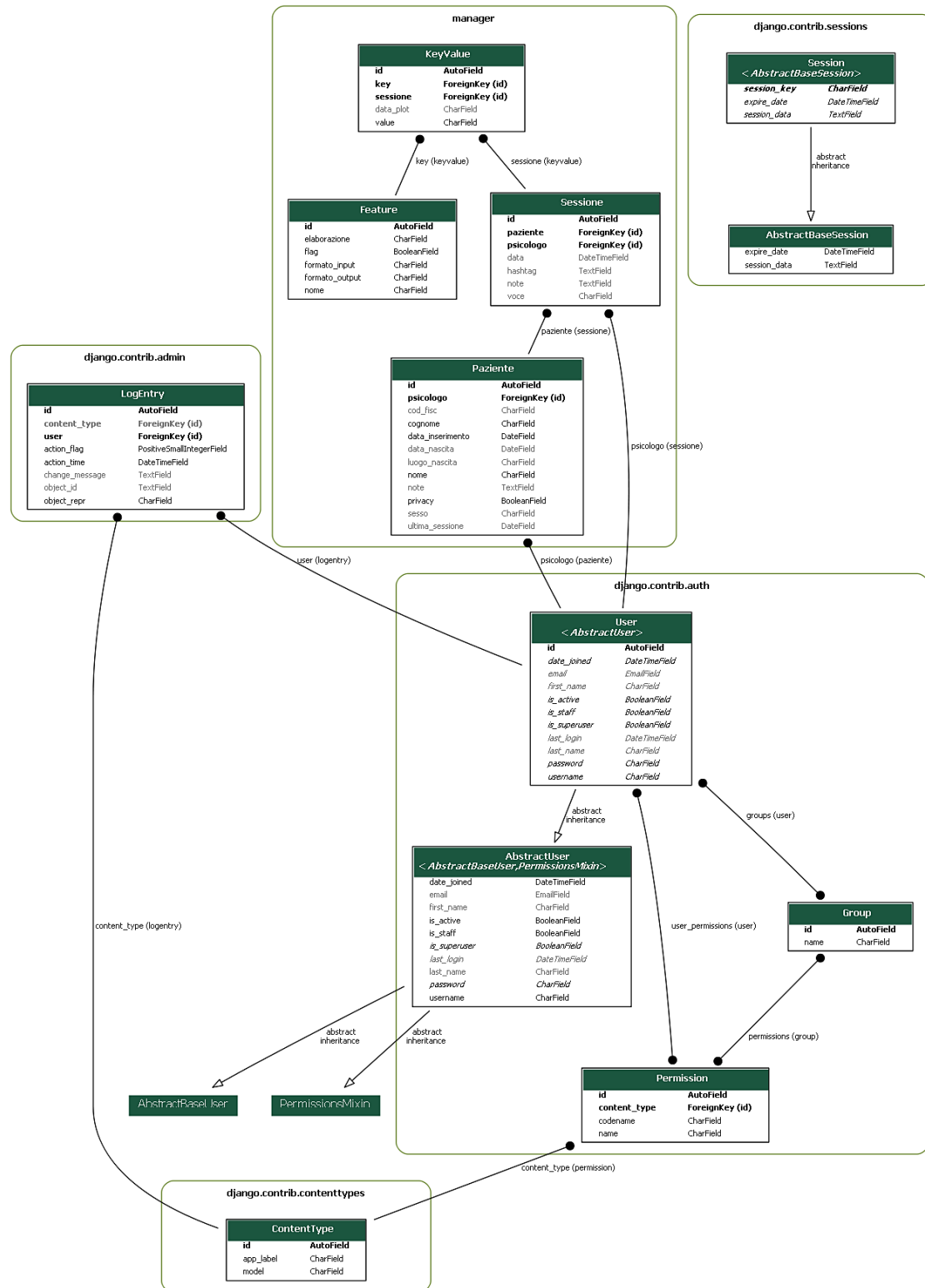


Figura 3.12 – Class Diagram di BrainIO. ¹¹

¹¹ Diagramma ottenuto utilizzando la libreria *pygraphviz*.

Quando si crea un progetto Django, nella directory di questo viene automaticamente inserita una cartella con il nome del progetto stesso, contenente il file di configurazione, *settings.py*.

In tale file saranno indicate le applicazioni Django create, i package utilizzati nel progetto, i template definiti, le variabili di ambiente, le configurazioni necessarie per la distribuzione su AWS, la posizione dei file statici e tutte le altre impostazioni del progetto.

Il progetto Django si compone delle seguenti applicazioni:

- *accounts*: gestisce la creazione, l'autenticazione, la modifica e l'eliminazione degli utenti, utilizzando il sistema di autenticazione di Django incluso nel package *django.contrib.auth*.

Per la registrazione di un nuovo utente, con i dati inseriti nell'apposito form, viene creata una nuova entry nella tabella *auth_user* del database, settando a *False* l'attributo booleano *is_active*, in modo da disabilitare temporaneamente l'account.

```
def registrazioneView(request):
    """
    View per la registrazione di un nuovo utente.
    """
    if request.method == 'POST':
        form = FormRegistrazione(request.POST)
        if form.is_valid():
            user = form.save(commit=False)
            user.is_active = False
            user.save()
            to_email = form.cleaned_data.get('email')

            sender(request, user, to_email)

            return render(request, 'accounts/mail_send.html')
        else:
            form = FormRegistrazione()
        return render(request, 'accounts/registrazione.html', {'form': form})
```

Viene, inoltre, inviata una mail all'indirizzo indicato in fase di registrazione per confermarlo tramite il one-time link contenente un token, generato ereditando dalla classe *PasswordResetTokenGenerator* contenuta in *django.contrib.auth.tokens*, e l'id dell'utente codificato in Base64.

La mail viene inviata istanziando la classe *EmailMessage* del package *django.core.mail* e passando come messaggio una stringa ottenuta effettuando il rendering del template *acc_active_email.html*, inserendo il token precedentemente generato nel *context*, il dizionario utilizzato dalle view di Django per passare dei valori ai template.

Premendo sul link ricevuto, si viene reindirizzati al template *registration_completed.html*, gestito dalla view *activate()*, la quale verifica che il token non sia stato alterato e recupera i dati dell'utente identificato dalla decodifica dell'id presente nell'URL. Infine, aggiorna la riga della tabella del database relativa all'utente interessato, settando a *True* il campo *is_active*.

Per l'invio della mail è necessario inserire nel file *settings.py* la configurazione del server del servizio di posta elettronica da utilizzare e, per una maggiore sicurezza, evitando di inserire le credenziali di accesso nel codice, è stato predisposto un file *dotenv*, *email.env*, nel quale inserire tali credenziali, recuperabili includendo il package *python-dotenv*.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_USE_TLS = True

# inserire nome del server SMTP del servizio di posta elettronica
EMAIL_HOST = ''
# inserire porta SMTP del servizio di posta elettronica
EMAIL_PORT = 587
# inserire username e password nel file email.env
EMAIL_HOST_USER = os.getenv('EMAIL_HOST_USER')
EMAIL_HOST_PASSWORD = os.getenv('EMAIL_HOST_PASSWORD')
DEFAULT_FROM_EMAIL = 'BrainIO Team '
```

Quando si elimina un utente dal database, prima di procedere con tale operazione, viene invocato il metodo `delete_folder()` per rimuovere definitivamente tutti i file delle elaborazioni delle sessioni presenti nell'apposito bucket di Amazon S3.

Questo viene invocato dal metodo `connect()` dell'istanza `pre_delete` della classe `ModelSignal`, la quale è a sua volta una sottoclasse di `Signal`.

Django include un “*signal dispatcher*” che consente alle applicazioni disaccoppiate di ricevere notifiche quando le azioni si verificano altrove nel framework.

I segnali consentono a determinati mittenti di notificare a una serie di destinatari che è stata eseguita un'azione. Sono particolarmente utili quando molte parti di codice possono essere interessate agli stessi eventi.

Si hanno due elementi chiave: il *receiver*, ovvero una funzione o un metodo di un'istanza che deve ricevere i segnali e, il *sender*, il quale se non definito, consente la ricezione di segnali da qualsiasi mittente. La connessione tra i due viene effettuata mediante il metodo `connect()` delle istanze di `Signal`.

Sono forniti diversi segnali built-in, come `django.db.models.signals.pre_delete`, inviato prima dell'eliminazione dell'istanza di un model Django, con il metodo `delete()`.

```
def delete_folder(sender, instance, **kwargs):
    if sender == User:
        lista = [f'sessioni/{instance.username}/']

        try:
            s3 = boto3.resource(
                's3', region_name=settings.AWS_S3_REGION_NAME)

            for path in lista:
                response = s3.meta.client.list_objects_v2(
                    Bucket=settings.AWS_STORAGE_BUCKET_NAME,
                    Prefix=path,
                )

                for element in response['Contents']:
                    s3.Object(
                        settings.AWS_STORAGE_BUCKET_NAME,
                        element['Key']
                    ).delete()
            except (boto3.client('s3').exceptions.NoSuchKey, KeyError) as e:
                pass
```

```
pre_delete.connect(delete_folder, sender=User)
```

- *core*: responsabile della visualizzazione della home page e della funzione di ricerca di una stringa inserita dall'utente tra le informazioni dei pazienti inseriti e delle relative sessioni.
- *data_list*: si occupa della visualizzazione dell'elenco e dei dettagli di pazienti e sessioni, mostrando i relativi grafici.

Per quanto concerne l'elenco dei pazienti inseriti, la paginazione dei dati è stata realizzata definendo una view class-based come sottoclasse di una view built-in, *ListView*, disponibile in *django.views.generic.list*. Tale view permette una paginazione della lista di oggetti mostrati, ciascuno dei quali identifica un paziente specifico, limitando il numero di oggetti per pagina tramite l'attributo *paginate_by*. Effettuando l'overriding del metodo *get_context_data()* è stato possibile ordinare gli oggetti della lista passando nel *context* due valori, *sorting* e *order_by*, con i quali viene specificato se ordinare, rispettivamente, in modo ascendente o discendente per nome, cognome, data di inserimento o ultima sessione. Questi vengono selezionati dall'utente tramite l'apposito dropdown definito nel template e recuperati dalla view nei parametri della richiesta HTTP, effettuata con il metodo GET.

Infine, ridefinendo il metodo *get_queryset()*, vengono recuperati gli oggetti del model *Paziente* in modo ordinato, a seconda dei valori indicati precedentemente.

Di default l'ordinamento è ascendente per cognome.

Per la paginazione della lista delle sessioni, accessibile solamente visualizzando i dettagli di uno specifico paziente, invece, è stata implementata una view function-based, nella quale viene creata un'istanza della classe *Paginator*, contenuta nel modulo *django.core.paginator*. Tale classe fornisce dei metodi che possono essere usati direttamente in un template Django per ottenere, ad esempio, il numero di pagine, la prima o l'ultima, quella precedente oppure quella successiva.

Ciascuna pagina consiste in una lista di un numero di oggetti definito nell'istanziamento della classe stessa, ognuno dei quali identifica una determinata sessione.

```

class PazientiListView(LoginRequiredMixin, ListView):
    """
    View per la visualizzazione dell'elenco dei pazienti inseriti dall'utente
    """
    context_object_name = 'elenco_pazienti'
    template_name = 'data_list/pazienti.html'
    paginate_by = 10

    def get_context_data(self, *args, **kwargs):
        context = super(
            PazientiListView, self).get_context_data(*args, **kwargs)

        sorting = 'ascendente'
        if self.request.GET.get('sorting') == 'discendente':
            sorting = 'discendente'

        order_by = 'cognome'
        if self.request.GET.get('order_by') is not None:
            order_by = self.request.GET.get('order_by')

        context['sorting'] = sorting
        context['order_by'] = order_by
        return context

    def get_queryset(self):
        order_by = 'cognome'
        if self.request.GET.get('order_by') is not None:
            order_by = self.request.GET.get('order_by')

        if self.request.GET.get('sorting') == 'discendente':
            order_by = '-' + order_by

        return Paziente.objects.filter(
            psicologo=self.request.user).order_by(order_by, 'cognome')

```

Per garantire l'accesso ad una view *function-based* solo se l'utente è autenticato, si utilizza il decoratore `@login_required`, mentre, per quelle *class-based* si eredita dalla classe `LoginRequiredMixin`.

Per realizzare i grafici da visualizzare nelle pagine di dettaglio dei pazienti e delle sessioni è stato utilizzato il package `matplotlib`, trasformando le figure che compongono tali grafici in HTML con il metodo `fig_to_html()`, incluso nel package `mpld3`.

- *manager*: gestisce la creazione, l'eliminazione e la modifica di un paziente o di una sessione.

In tale applicazione, nel file *models.py*, sono definiti i model della web application, i quali vengono mappati in tabelle, come precedentemente descritto.

Il database di BrainIO è costituito da cinque tabelle principali, trascurando quelle in relazione con *auth_user*, associate a model Django già preconfigurati.

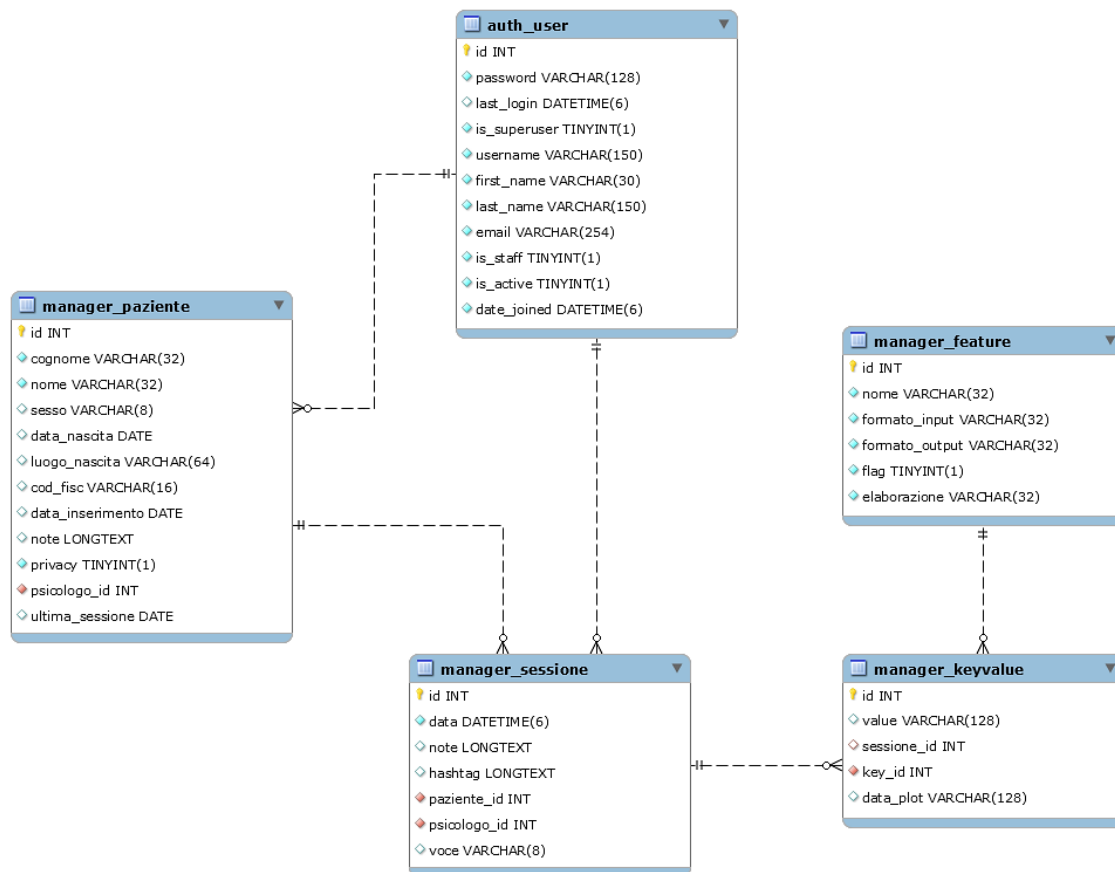


Figura 3.13 – Struttura database di BrainIO.

Ad ogni utente (tabella *auth_user*) possono essere associati zero o più pazienti (tabella *manager_paziente*) e zero o più sessioni (tabella *manager_sessione*).

Ciascun paziente può essere associato a zero o più sessioni, le quali, a loro volta, sono associate a zero o più keyvalue (tabella *managaer_keyvalue*).

La tabella keyvalue è utilizzata per mantenere un riferimento tra le feature e i file risultanti dalle elaborazioni sui video, i quali sono memorizzati in un bucket di Amazon S3.

Infatti, a ciascun keyvalue è associato una ed una sola feature, viceversa, ad una feature possono essere associati zero o più keyvalue.

Ciascun campo di una tabella coincide con un attributo di un model, il quale può essere di diversi tipi e viene definito come istanza di un'appropriata sottoclasse di *Field*. Tra quelli più comunemente utilizzati troviamo *CharField*, usato per definire stringhe di lunghezza fissa di dimensioni medio-piccole, *TextField*, per stringhe di lunghezza arbitraria di grandi dimensioni, *IntegerField*, per valori interi, *DateField* e *DateTimeField*, *EmailField*, *FileField* e *ImageField*, per l'upload rispettivamente di file e immagini, *ForeignKey*, per specificare una relazione uno-a-molti con un altro model e *ManyToManyField*, per specificare una relazione multi-a-molti.

Per ciascun campo è possibile definire delle opzioni, come ad esempio *null*, per consentire la memorizzazione di valori nulli, *blank*, il quale, se settato a *False*, costringe l'utente ad inserire un valore nel campo in fase di compilazione del relativo form e *unique*, per specificare che quel campo deve essere unico per tutta la tabella.

Inoltre, una chiave primaria può essere dichiarata settando a *True* l'apposita opzione *primary_key* e, se per nessun campo di un model dovesse essere definita tale opzione, allora Django crea automaticamente una chiave primaria *auto-increment*.

Configurando l'opzione *choices* con una lista di tuple, il widget HTML di default, ovvero la rappresentazione di Django di un elemento di input HTML, sarà un select box, anziché un text field standard.

Il primo elemento in ciascuna tupla è il valore che sarà memorizzato nel database, mentre l'altro sarà mostrato tra le scelte nel select box.

Ad esempio, nel model *Paziente* tale opzione è stata utilizzata per specificare il sesso dell'individuo.

La classe *Meta* consente la dichiarazione di metadati per un model. L'attributo *ordering* è utile per impostare una lista di campi per i quali effettuare l'ordinamento predefinito dei record quando si effettua una query sul model. Di default l'ordinamento è in senso ascendente, ma antepoendo il simbolo '-' può essere invertito.

Infine, con l'attributo *verbose_name* si definisce un nome dettagliato per la classe in forma singolare e plurale.

Il metodo `__str__()` restituisce una rappresentazione testuale di un oggetto e, `get_absolute_url()` restituisce un URL per accedere ad una particolare istanza di un model.

```
class Paziente(models.Model):
    GENDER_CHOICES = (
        ('M', 'Maschio'),
        ('F', 'Femmina')
    )
    cognome = models.CharField(max_length=32)
    nome = models.CharField(max_length=32)
    sesso = models.CharField(choices=GENDER_CHOICES,
                             max_length=8, null=True, blank=True)
    data_nascita = models.DateField(null=True, blank=True)
    psicologo = models.ForeignKey(User, on_delete=models.CASCADE)

    ...

class Meta:
    ordering = ['cognome']
    verbose_name = 'paziente'
    verbose_name_plural = 'pazienti'

def __str__(self):
    return f'{self.cognome} {self.nome} - C.F. {self.cod_fisc}'

def get_absolute_url(self):
    return reverse('dettagli_paziente_view', kwargs={'pk': self.pk})
```

Nel file `models.py` sono stati configurati due segnali built-in Django, `django.db.models.signals.post_delete`, in modo tale che, in seguito all'eliminazione di un paziente o di una sessione, venga invocato il metodo `delete_resorces()`, il quale si occupa di rimuovere definitivamente i file ad essi associati, dal bucket di Amazon S3.

Nella sottocartella `fixtures` è stato inserito un file JSON, `initial_features.json`, il quale contiene i record della tabella `manager_feature`, caricati automaticamente in fase di inizializzazione del progetto Django.

Ciascun record identifica una *feature* realizzata, specificando il nome dettagliato di quest'ultima, il formato del file accettato in input e quello dell'output, l'elaborazione effettuata e un flag, che indica se la feature deve essere elaborata in fase di caricamento del video.

Per l'inserimento di una nuova sessione, dopo aver recuperato e verificato i dati inseriti dall'utente nel form, viene controllato, innanzitutto, che non ne esista già un'altra dello stesso paziente, alla stessa ora.

Il video caricato deve rispettare alcuni criteri per poter essere fornito in input al servizio Amazon Transcribe, il quale andrà a trascriverlo in un formato testuale.

Viene verificata che la dimensione massima del file sia di 2 GB e, che la sua estensione sia tra quelle supportate (m4v, mov, mp4, ogm, ogv, ogx, webm).

Prima di caricare il video nel bucket di S3 riservato alla memorizzazione di tali contenuti multimediali, *brainio-mediatemp*, il file viene rinominato inserendo il prefisso *'brainiovideo_'*, il quale sarà specificato poi nel trigger della funzione Lambda *aws-start-processing*, seguito dall'id dello psicologo e del paziente e, dalla data e l'ora della sessione.

Una volta completato l'upload del video, viene creata la nuova entry della tabella *manager_sessione* e, quindi, una nuova istanza del model *Sessione*.

Infine, viene aggiornato il campo *ultima_sessione* dell'istanza del model *Paziente*, associata alla sessione inserita.

```
@login_required
def nuovaSessione(request, paziente):
    """
    Funzione per la creazione di una nuova sessione e caricamento video
    """
    form = FormSessione(request.user, request.POST, request.FILES)
    if form.is_valid():
        data = form.cleaned_data['data']
        note = form.cleaned_data.get('note')
        hashtag = getHashtags(form.cleaned_data['hashtag'])

        psicologo = request.user

        if Sessione.objects.filter(
            psicologo=psicologo, paziente=paziente, data=data
        ).count() > 0:
            messages.error(
                request,
                f'ERRORE: Esiste già una sessione del paziente {paziente} \
                allo stesso orario')
        return
```

```

file = request.FILES['file']
# controlla che la dimensione del file sia al massimo di 2GB
if not file.size <= 2147483648:
    messages.error(
        request, 'ERRORE: La dimensione massima consentita del \
            video è di 2GB')
    return

# controlla l'estensione del file caricato
extension_supported = [
    'm4v', 'mov', 'mp4', 'ogm', 'ogv', 'ogx', 'webm']
formato_file = file.name.split('.').pop()
if formato_file.lower() not in extension_supported:
    messages.error(
        request, 'ERRORE: Il formato del video inserito non è \
            supportato. Formati supportati: \'m4v\', \'mov\', \
            \'mp4\', \'ogm\', \'ogv\', \'ogx\', \'webm\'')
    return

try:
    s3 = boto3.resource(
        's3', region_name=settings.AWS_S3_REGION_NAME)
    bucket = s3.Bucket(settings.AWS_VIDEO_BUCKET_NAME)
    filename = 'brainiovideo_' + str(psicologo.pk) + '_' + str(
        paziente.pk) + '_' + str(data).replace(':', '-').replace(
            ' ', '_') + '.' + formato_file

    bucket.put_object(Key=filename, Body=file)

    # controlla che il video sia stato caricato correttamente
    if bucket.objects.filter(Prefix=filename):
        messages.success(
            request, 'OK: Video caricato correttamente')
    else:
        messages.error(request, 'ERRORE: Caricamento video non \
            riuscito')
        return

    sessione = Sessione.objects.create(
        paziente=paziente,
        data=data,
        note=note,
        hashtag=hashtag,
        psicologo=psicologo
    )

    # controlla che la nuova entry sia stata inserita correttamente
    if Sessione.objects.filter(pk=sessione.pk).count() > 0:
        messages.success(request, 'OK: Nuova sessione inserita \
            correttamente')
        aggiornaUltimaSessione(paziente)

```

```

else:
    if bucket.objects.filter(Prefix=filename):
        bucket.objects.filter(Prefix=filename).delete()

    messages.error(request, 'ERRORE: Non è stato possibile \
        creare la sessione')
    return

except IntegrityError:
    messages.error(request, 'ERRORE: Non è stato possibile \
        creare la sessione')
    return
except (boto3.client('s3').exceptions.NoSuchKey, KeyError) as e:
    pass
else:
    messages.error(request, 'ERRORE: Inserimento nuova sessione non \
        riuscito')

```

In generale per poter visualizzare, modificare o eliminare una sessione è necessario attendere il completamento di tutte le elaborazioni sul video caricato, altrimenti potrebbero andare persi gli output delle feature, oppure generati file non necessari. Questo viene verificato mediante la funzione *isProcessingVideo()*, definita nel file *views.py* dell'applicazione *manager* e, importata in tutte le altre applicazione che richiedono un'interazione con un'istanza del model *Sessione*.

Quando si accede all'URL per la modifica di una sessione, la view invocata passa al template i valori per preconfigurare il form con le informazioni della sessione da modificare ed una lista con i segmenti della trascrizione del dialogo tra il paziente e lo psicologo, contenuti nell'output di Amazon Transcribe e formattato adeguatamente in un file con suffisso *'_transcribe-speaker.json'*.

```

@login_required
def modificaSessioneView(request, pk):
    """
    Classe per la modifica di una sessione
    """
    sessione_old = Sessione.objects.get(pk=pk)
    context = {}

    if isProcessingVideo(
        request.user, sessione_old.paziente, sessione_old.data):
        messages.warning(request, f'ATTENZIONE: Elaborazione video \
            "{sessione_old}" ancora in corso! Non è possibile procedere alla \
            modifica della sessione')

```

```

        return HttpResponseRedirect(sessione_old.get_absolute_url())

    if request.method == 'POST':
        form = FormModificaSessione(
            request.user, request.POST)

        if form.is_valid():
            paziente = form.cleaned_data['paziente']
            data = form.cleaned_data['data']
            voce = request.POST.get('voce')
            note = form.cleaned_data.get('note')
            hashtag = getHashtags(form.cleaned_data['hashtag'])
            psicologo = request.user

            try:
                if Sessione.objects.filter(
                    psicologo=psicologo,
                    paziente=paziente,
                    data=data
                ).exclude(pk=sessione_old.pk).count() > 0:
                    messages.error(
                        request,
                        f'ERRORE: Esiste già una sessione del paziente \
                        {paziente} allo stesso orario')
                    return HttpResponseRedirect(
                        sessione_old.get_absolute_url()
                    )

                sessione_new = moveSessionResources(
                    sessione_old,
                    paziente,
                    data,
                    voce,
                    note,
                    hashtag
                )

                aggiornaUltimaSessione(paziente)
                messages.success(request, 'OK: Sessione modificata \
                correttamente.')
                return HttpResponseRedirect(sessione_new.get_absolute_url())
            except IntegrityError:
                messages.error(request, 'ERRORE: Non è stato possibile \
                modificare la sessione')
                return HttpResponseRedirect(sessione_old.get_absolute_url())
        else:
            form = FormModificaSessione(
                request.user,
                initial={
                    'data': sessione_old.data,
                    'paziente': sessione_old.paziente_id,
                    'note': sessione_old.note,
                    'hashtag': sessione_old.hashtag
                }
            )

```

```

context = {
    'transcribe': {
        'voce_1': [],
        'voce_2': [],
        'initial_voice': sessione_old.voce
    }
}

try:
    s3 = boto3.client('s3', region_name=settings.AWS_S3_REGION_NAME)
    response = s3.list_objects_v2(
        Bucket=settings.AWS_STORAGE_BUCKET_NAME,
        Prefix=getPathSessioni(
            request.user.pk,
            sessione_old.paziente.pk,
            sessione_old.data
        ) + str(sessione_old.data).replace(':', '-').replace(
            ' ', '_') + '_transcribe-speaker.json'
    )

    if response['KeyCount'] == 1:
        file_obj = s3.get_object(
            Bucket=settings.AWS_STORAGE_BUCKET_NAME,
            Key=getPathSessioni(
                request.user.pk,
                sessione_old.paziente.pk,
                sessione_old.data
            ) + str(sessione_old.data).replace(':', '-').replace(
                ' ', '_') + '_transcribe-speaker.json'
        )

        transcript_result = json.loads(
            file_obj['Body'].read()
        )

        for segment in transcript_result['segments']:
            if segment['speaker'] == 'Voce_1':
                context['transcribe']['voce_1'].append(
                    segment['content']
                )
            else:
                context['transcribe']['voce_2'].append(
                    segment['content']
                )

except (boto3.client('s3').exceptions.NoSuchKey, KeyError) as e:
    pass

context['form'] = form
return render(request, 'manager/modifica_sessione.html', context)

```

Nella view per la modifica della sessione, oltre a verificare la validità dei dati inseriti dall'utente e, la non esistenza di un'altra sessione dello stesso paziente alla stessa ora, vengono passati tali dati ad un'altra funzione, *moveSessionResources()*, la quale andrà a “*spostare*” i file relativi alle elaborazioni nel bucket di Amazon S3, ovvero modifica i prefissi dei nomi di chiave e le chiavi stesse degli oggetti nel bucket.

Tale funzione, inoltre, inserisce un nuovo record nella tabella *manager_sessione*, andando ad eliminare quello relativo alla sessione modificata.

In seguito alla modifica della chiave dei file JSON nel bucket *brainiostorage*, viene invocata la funzione Lambda *aws-end-processing*, la quale inserirà i record contenenti i riferimenti ai file di ciascuna feature, nella tabella *manager_keyvalue* del database.

```
def moveSessionResources(sessione_old, paziente, data, voce, note, hashtag):
    ...

    try:
        s3 = boto3.resource('s3', region_name=settings.AWS_S3_REGION_NAME)

        src_path = getPathSessioni(
            sessione_old.psicologo.pk,
            sessione_old.paziente.pk,
            sessione_old.data
        )

        dest_path = getPathSessioni(
            sessione_old.psicologo.pk,
            paziente.pk,
            data
        )

        response = s3.meta.client.list_objects_v2(
            Bucket=settings.AWS_STORAGE_BUCKET_NAME,
            Prefix=src_path
        )

        for element in response['Contents']:
            copy_source = {
                'Bucket': settings.AWS_STORAGE_BUCKET_NAME,
                'Key': element['Key']
            }

            s3.Object(
                settings.AWS_STORAGE_BUCKET_NAME,
                dest_path + dest_path.split('/')[ -2] + '_' +
                element['Key'].split('/')[ -1].split('_')[ -1]
            ).copy_from(CopySource=copy_source)

    ...
```

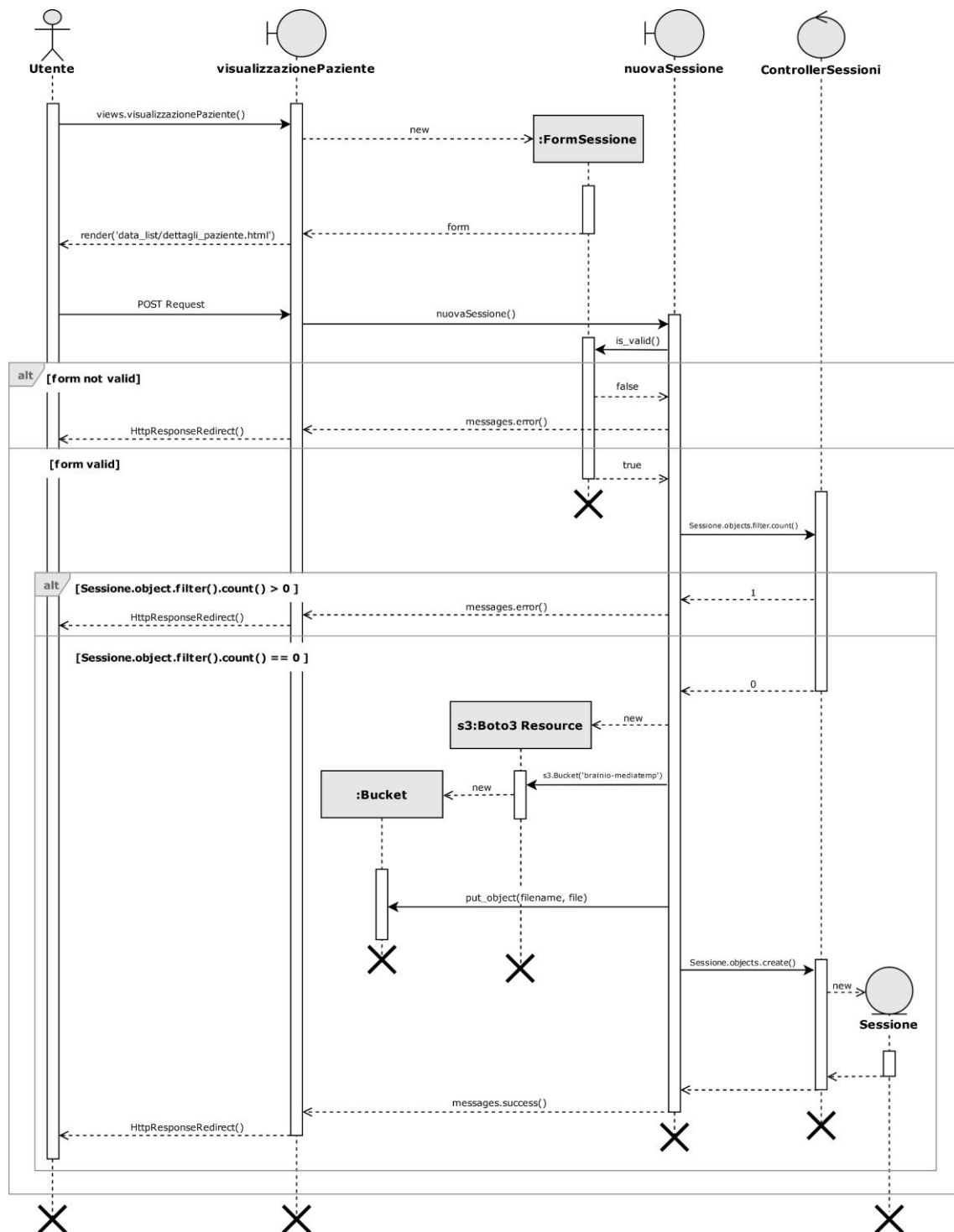


Figura 3.14 – Sequence Diagram della creazione di una nuova sessione.

3.3 Distribuzione su AWS e funzioni Lambda

Per configurare l'interfaccia a riga di comando di Elastic Beanstalk, utilizzata per creare un ambiente e distribuire una versione dell'applicazione, è stato definito un utente IAM, inserendo le credenziali nel file di configurazione memorizzato in locale.

Una volta creata un'applicazione Elastic Beanstalk, specificando il nome, una regione AWS e il linguaggio di programmazione utilizzato, è necessario creare un ambiente, il cui nome di dominio (*CNAME*) deve essere inserito nella lista degli *ALLOWED_HOSTS* nel file *settings.py* del progetto Django.

Dopo ogni modifica del codice del progetto in locale, questo può essere distribuito mediante il comando *eb deploy* dell'interfaccia a riga di comando *eb-cli*. Tale comando crea una nuova versione dell'applicazione, la quale diventa automaticamente la versione in esecuzione nell'ambiente precedentemente definito.

Dalla console di gestione di Elastic Beanstalk è possibile visualizzare l'elenco delle versioni dell'applicazione e recuperarne una in caso di problemi.

Per la memorizzazione dei dati necessari per il corretto funzionamento della web application sono stati definiti tre bucket di Amazon S3:

- *brainiostorage*: è il bucket principale dell'applicazione, utilizzato per salvare i file ottenuti dalle elaborazioni sui video caricati.
- *brainio-mediatemp*: usato per memorizzare temporaneamente il video di una sessione, il quale viene eliminato al completamento delle elaborazioni.
- *brainio-resulttemp*: impiegato dalla funzione Lambda *aws-start-processing*, per salvare i file temporanei ottenuti in output dalle elaborazioni di Amazon Transcribe.

Il bucket *brainiostorage* viene configurato in modo da bloccare l'accesso pubblico ai suoi oggetti, garantendo l'accesso ad essi solamente all'ambiente Elastic Beanstalk. Questo viene effettuato definendo una policy del bucket e specificando come elemento *Principal* l'ARN del ruolo IAM associato all'istanza EC2.

In tal modo si evita di inserire le credenziali di accesso di un utente IAM nel codice del progetto Django.


```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::696482742108:role/aws-elasticbeanstalk-ec2-role"
      },
      "Action": [
        "s3:ListBucket",
        "s3:ListBucketVersions",
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:DeleteObject",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::brainiostorage",
        "arn:aws:s3:::brainiostorage/*"
      ]
    }
  ]
}

```

Un'istanza database di Amazon RDS può essere configurata direttamente dalla console di gestione di Elastic Beanstalk, accedendo alle impostazioni dell'ambiente.

Per BrainIO è stato scelto un motore di database *mysql*, di versione *8.0.21*, una classe di istanza *db.t2.micro* ed uno spazio di memorizzazione di 5 GB.

Inoltre, è possibile definire nome utente e password per accedere all'istanza, i quali, insieme al nome del database, alla porta e all'endpoint sono stati inseriti tra le proprietà dell'ambiente Elastic Beanstalk, per poterle passare all'applicazione, includendole nel file *settings.py*, al fine di connettere la web application al database.

Il progetto Django è stato configurato in modo tale che utilizzi l'istanza database di Amazon RDS, se tali proprietà sono state definite tra le variabili di ambiente, altrimenti verrà creato automaticamente un database *SQLite* in locale.

```

if 'RDS_HOSTNAME' in os.environ:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.mysql',
            'NAME': os.environ['RDS_DB_NAME'],
            'USER': os.environ['RDS_USERNAME'],
            'PASSWORD': os.environ['RDS_PASSWORD'],
            'HOST': os.environ['RDS_HOSTNAME'],
            'PORT': os.environ['RDS_PORT'],
        }
    }
else:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.sqlite3',
            'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
        }
    }

```

Per consentire il traffico in entrata sulla porta appropriata è necessario configurare il gruppo di sicurezza associato all'istanza database. Per fare ciò basta accedere alla scheda *Connectivity & security* della pagina di visualizzazione dei dettagli dell'istanza, nella console di gestione di Amazon RDS e premere sull'apposito link per visualizzare il gruppo di sicurezza nella console di Amazon EC2. Da qui è possibile aggiungere una nuova regola in entrata, specificando un numero di porta o un intervallo di porte, come tipo, il motore di database utilizzato dall'applicazione e, come origine, il nome del gruppo di sicurezza associato all'ambiente Elastic Beanstalk.

In tal modo si consente alle istanze EC2 dell'ambiente di accedere al database.

Per le elaborazioni sui video delle sessioni sono state definite tre funzioni Lambda che utilizzano i due servizi di AWS precedentemente descritti, Amazon Transcribe e Amazon Comprehend.

Tali funzioni, oltre ai vantaggi relativi al serverless computing, sono completamente disaccoppiate dal resto del codice del progetto Django, consentendo di continuare ad utilizzare la web application durante le elaborazioni, senza alterare le sue prestazioni.

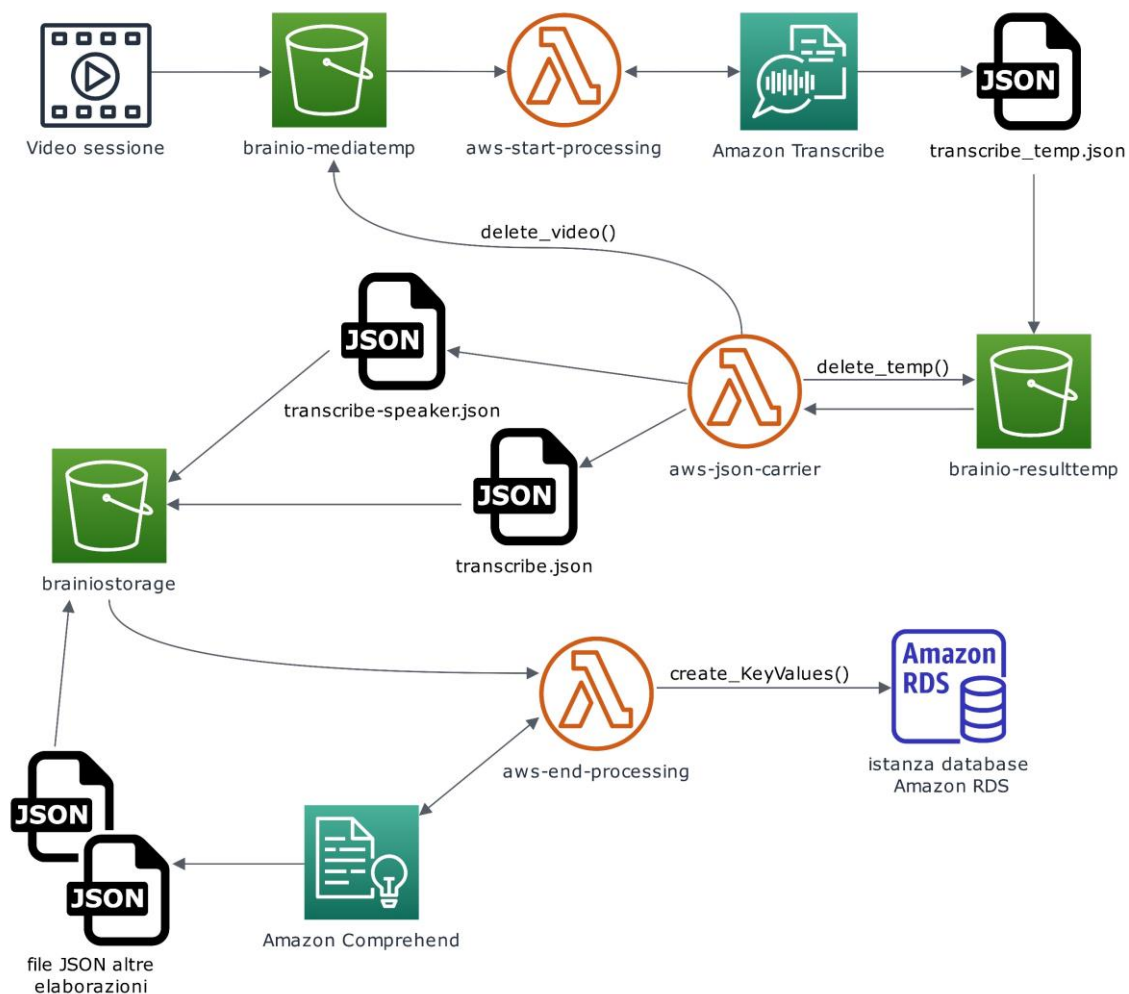


Figura 3.15 – Il diagramma illustra la sequenza di invocazione delle tre funzioni Lambda.

A ciascuna funzione Lambda deve essere associato un ruolo IAM per concederle l'autorizzazione ad accedere ad altri servizi e alle risorse AWS.

Alla prima funzione Lambda invocata, *aws-start-processing*, è associato un ruolo che le consente un accesso completo ad Amazon S3 (*AmazonS3FullAccess*), Amazon Transcribe (*AmazonTranscribeFullAccess*) e, naturalmente, alle funzionalità di esecuzione di AWS Lambda (*AWSLambdaExecute*).

Questa viene attivata in seguito a “*tutti gli eventi di creazione oggetti*” (sono incluse le operazioni di PUT, POST, COPY, caricamento a più parti) con prefisso “*brainiovideo_*”, nel bucket *brainio-mediatemp* di Amazon S3.

Tale funzione effettua, mediante il servizio Amazon Transcribe, la trascrizione del dialogo estratto dal video caricato. Quest'ultima viene avviata con il metodo

start_transcription_job() del client di basso livello fornito dall'SDK di AWS per Python, *Boto3*, passando come parametri il nome del file di output e del bucket in cui memorizzarlo (*brainio-resulttemp*), l'URI, il formato e la lingua del video in input.

Inoltre, tra le impostazioni opzionali, viene settato *ShowSpeakerLabels* a *True*, il quale abilita l'identificazione degli oratori, etichettando opportunamente ciascuno di essi nel file JSON risultante e, viene specificata in *MaxSpeakerLabels* la presenza di massimo due oratori.

L'output di tale elaborazione conterrà l'intera trascrizione del dialogo e una lista di segmenti ottenuta dall'alternanza degli interlocutori, nella quale ciascun segmento contiene, a sua volta, una lista di elementi che identificano le parole nel dialogo, senza mostrarne il contenuto, per le quali viene specificata l'etichetta associata all'oratore e il tempo di inizio e di fine in cui tali parole sono state pronunciate all'interno video.

Inoltre, viene prodotta un'ulteriore lista delle parole individuate nel dialogo. Tale lista, a differenza della precedente, mostra il contenuto di queste e specifica per ognuna un valore compreso tra 0 e 1, che rappresenta il punteggio di affidabilità dell'identificazione.

L'operazione di *PUT* di un file con suffisso *.json* nel bucket *brainio-resulttemp*, costituisce l'evento trigger della funzione Lambda *aws-json-carrier*, alla quale viene associato un ruolo che le garantisce un accesso, oltre che alle funzionalità di esecuzione di AWS Lambda, anche ad Amazon S3.

In tale funzione viene prodotto un file JSON a partire da quello risultante dall'elaborazione precedente, trasformandolo in un formato facilmente leggibile e utilizzabile, successivamente, da Amazon Comprehend.

In pratica, vengono unite le due liste presenti nel file originale, generando un'unica lista di segmenti, nella quale viene riportato, per ciascuno di essi, oltre al tempo di inizio e di fine all'interno del video e l'etichetta associata ad uno specifico oratore, anche il contenuto per intero, ossia le frasi pronunciate prima che un interlocutore venga interrotto dall'altro. Questo nuovo file viene inserito nel bucket *brainiostorage*, insieme al file originale, il quale viene eliminato dal bucket nel quale era stato memorizzato in precedenza.

Infine, tale funzione elimina il video caricato nel bucket *brainio-mediatemp*, in quanto non più necessario per le successive elaborazioni.

L'ultima funzione Lambda invocata è *aws-end-processing*, alla quale viene associato un ruolo che le concede le stesse autorizzazioni della funzione precedente e, in aggiunta, un accesso completo ad Amazon RDS (*AmazonRDSFullAccess*) e Amazon Comprehend (*ComprehendFullAccess*).

Questa viene attivata in seguito a “*tutti gli eventi di creazione oggetti*” con prefisso “*sessioni*” e suffisso “.json” nel bucket *brainiostorage*.

Per ciascun file JSON caricato in tale bucket, se si tratta del file prodotto dall'elaborazione precedente (*transcribe-speaker.json*), allora vengono avviate le analisi dei sentimenti (*batch_detect_sentiment()*) e delle frasi chiave (*batch_detect_key_phrases()*), passando come parametro il contenuto di tale file.

Inoltre, vengono creati i file “*-axis*” per le elaborazioni che devono essere rappresentate con un grafico nella pagina di dettaglio di un paziente o di una sessione e, vengono inseriti, nella tabella *manager_keyvalue*, i riferimenti ai file prodotti dalle feature.

Questa funzione Lambda viene invocata anche in seguito ad una modifica della data e l'ora di una sessione, oppure del paziente ad essa associato. Infatti, in tali casi si effettua un'operazione di *COPY* dei file JSON nel bucket principale della web application ed è, quindi, necessario inserire nuovi record nella tabella del database.

Conclusioni

Lo sviluppo della web application mi ha permesso di acquisire nuove conoscenze, a partire dal linguaggio di programmazione Python, consolidando quelle già apprese durante il percorso universitario e assumendo una maggiore consapevolezza dell'importanza delle nozioni trattate nei vari insegnamenti.

Sarebbe necessario, tuttavia, un approfondimento sulle norme a tutela dei dati personali, nella gestione delle informazioni dei pazienti e nella memorizzazione, seppur temporanea, delle registrazioni video delle sessioni.

Per quanto concerne la progettazione dell'applicazione, una maggiore esperienza con la vasta gamma di servizi di cloud computing offerti da AWS potrebbe portare ad un'ottimale configurazione dei servizi scelti, in particolar modo in ambito di sicurezza.

Al fine di migliorare ulteriormente la modularità dell'applicazione, semplificando notevolmente l'integrazione di nuove feature, è possibile modificare le funzioni Lambda, in modo da realizzare la seguente architettura:

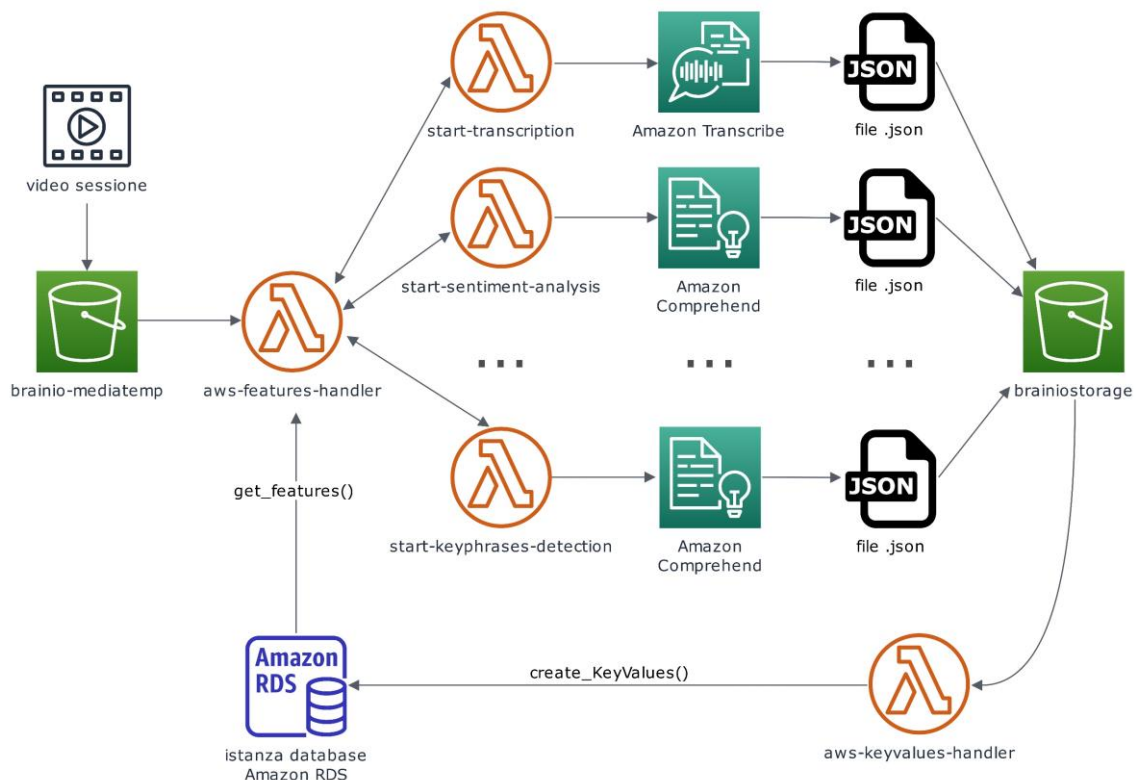


Figura 4.1 – Possibile miglioramento nella sequenza di invocazione delle funzioni Lambda.

Definire una funzione Lambda per ciascuna feature, invocandola direttamente dall'interno di un'altra funzione Lambda, permette di ridurre notevolmente alcune complessità legate all'implementazione di nuove elaborazioni da effettuare sui video delle sessione, le quali potrebbero utilizzare non necessariamente API di servizi di AWS, ma anche altre di terze parti.

Una volta scritto il codice della funzione Lambda relativa ad una specifica feature, è possibile inserire nel file *initial_features.json*, oltre ai dati che comporranno i record della tabella *manager_feature* del database, come previsto nella configurazione attuale, anche il nome della funzione Lambda da invocare.

```
[
  {
    "model": "manager.feature",
    "pk": 1,
    "fields": {
      "nome": "start-sentiment-analysis",
      "formato_input": "json",
      "formato_output": "json",
      "elaborazione": "sentiment",
      "flag": True
    }
  },
  ...
]
```

In tal modo, nella funzione *aws-features-handler*, configurata con lo stesso evento trigger della funzione *aws-start-processing*, ossia “*tutti gli eventi di creazione oggetti*” con prefisso “*brainiovideo_*” nel bucket *brainio-mediatemp*, è possibile accedere all'istanza database di Amazon RDS associata all'ambiente Elastic Beanstalk per recuperare la lista delle feature e, quindi, le funzioni Lambda da invocare.

Queste possono essere attivate direttamente all'interno della prima funzione Lambda invocata, mediante il metodo *invoke()* del client fornito da Boto3.

```

import boto3

client = boto3.client('lambda')

for feature in query_result:
    client.invoke(
        FunctionName=feature,
        InvocationType='RequestResponse'
    )

```

Infine, specificando come evento trigger dell'ultima funzione Lambda, *aws-keyvalues-handler*, “tutti gli eventi di creazione oggetti” con prefisso “*sessioni*” e suffisso “.json” nel bucket *brainiostorage*, questa si occuperà dell'inserimento degli opportuni record nella tabella *manager_keyvalue*.

In tal modo, come in precedenza, questa verrà invocata anche in seguito ad una modifica della data e l'ora di una sessione oppure del paziente ad essa associato.

A conclusione di questo elaborato, desidero ringraziare il mio relatore, il Professor Nicola Bicocchi, il quale mi ha proposto tale attività progettuale, supportandomi, con la sua infinita disponibilità ed esperienza, in ogni fase della realizzazione di questa, permettendomi di accrescere le mie conoscenze e competenze.

Riferimenti

- NIGEL GEORGE, 2019. *Build a Website With Django 3: A complete introduction to Django 3*. Hamilton NSW, Australia: GNW Independent Publishing.
- DAN C. MARINESCU, 2017. *Cloud Computing: Theory and Practice*. Cambridge, MA, United States: Morgan Kaufmann.
- NIST CLOUD COMPUTING STANDARDS ROADMAP WORKING GROUP, 2017. *NIST Cloud Computing Standards Roadmap*. Special Publication 500-291, Version 2.
- THOMAS ERL, RICARDO PUTTINI, ZAIGHAM MAHMOOD, 2013. *Cloud computing. Concepts, technology & architecture*. Westford, Massachusetts, United States: Prentice Hall.
- JÖRG KRAUSE, 2020. *Introducing Bootstrap 4: Create Powerful Web Applications Using Bootstrap 4.5*. (s.l.): Apress.
- SCOTT PATTERSON, 2019. *Learn AWS Serverless Computing: A beginner's guide to using AWS Lambda, Amazon API Gateway, and services from Amazon Web Services*. Birmingham, UK: Packt Publishing.
- *Python 3.6.13 documentation*. [online].
Disponibile su <<https://docs.python.org/3.6/>> [Data di accesso: marzo 2021].
- *Django documentation*. [online].
Disponibile su <<https://docs.djangoproject.com/en/2.1/>> [Data di accesso: marzo 2021].
- *Bootstrap documentation*. [online].
Disponibile su <<https://getbootstrap.com/docs/5.0/getting-started/introduction/>> [Data di accesso: marzo 2021].
- *Cos'è il cloud computing?* [online].
Disponibile su <<https://aws.amazon.com/it/what-is-cloud-computing/>> [Data di accesso: marzo 2021].
- *Infrastruttura globale*. [online].
Disponibile su <<https://aws.amazon.com/it/about-aws/global-infrastructure/?pg=WIAWS>> [Data di accesso: marzo 2021].
- *Principi fondamentali di AWS*. [online].

- Disponibile su <<https://aws.amazon.com/it/getting-started/fundamentals-core-concepts/?e=gs2020&p=gsr>> [Data di accesso: marzo 2021].
- *AWS Well-Architected*. [online].
Disponibile su <<https://aws.amazon.com/it/architecture/well-architected/?e=gs2020&p=fundcore&wa-lens-whitepapers.sort-by=item.additionalFields.sortDate&wa-lens-whitepapers.sort-order=desc>> [Data di accesso: marzo 2021].
 - *Modello di responsabilità condivisa*. [online].
Disponibile su <<https://aws.amazon.com/it/compliance/shared-responsibility-model/>> [Data di accesso: marzo 2021].
 - *Cosa sono i microservizi?* [online].
Disponibile su <<https://aws.amazon.com/it/microservices/>> [Data di accesso: marzo 2021].
 - *Serverless in AWS*. [online].
Disponibile su <<https://aws.amazon.com/it/serverless/>> [Data di accesso: marzo 2021].
 - *Icone per l'architettura di AWS*. [online].
Disponibile su <<https://aws.amazon.com/it/architecture/icons/>> [Data di accesso: marzo 2021].
 - *Documentazione AWS*. [online].
Disponibile su <https://docs.aws.amazon.com/it_it/index.html?nc2=h_ql_doc_do_v> [Data di accesso: marzo 2021].
 - *Documentazione su AWS Identity and Access Management*. [online].
Disponibile su <https://docs.aws.amazon.com/it_it/iam/?id=docs_gateway> [Data di accesso: marzo 2021].
 - *Documentazione su AWS Elastic Beanstalk*. [online].
Disponibile su <https://docs.aws.amazon.com/it_it/elastic-beanstalk/?id=docs_gateway> [Data di accesso: marzo 2021].
 - *Documentazione su Amazon Elastic Compute Cloud*. [online].
Disponibile su <https://docs.aws.amazon.com/it_it/ec2/?id=docs_gateway> [Data di accesso: marzo 2021].
 - *Documentazione su Amazon Simple Storage Service*. [online].

- Disponibile su <https://docs.aws.amazon.com/it_it/s3/?id=docs_gateway> [Data di accesso: marzo 2021].
- *Documentazione su Amazon Relational Database Service*. [online].
Disponibile su <https://docs.aws.amazon.com/it_it/rds/?id=docs_gateway> [Data di accesso: marzo 2021].
 - *Documentazione su AWS Lambda*. [online].
Disponibile su <https://docs.aws.amazon.com/it_it/lambda/?id=docs_gateway> [Data di accesso: marzo 2021].
 - *Documentazione su Amazon Transcribe*. [online].
Disponibile su <https://docs.aws.amazon.com/it_it/transcribe/?id=docs_gateway> [Data di accesso: marzo 2021].
 - *Documentazione su Amazon Comprehend*. [online].
Disponibile su <https://docs.aws.amazon.com/it_it/comprehend/?id=docs_gateway> [Data di accesso: marzo 2021].
 - *Boto3 documentation*. [online].
Disponibile su <<https://boto3.amazonaws.com/v1/documentation/api/latest/index.html>> [Data di accesso: marzo 2021].
 - *Serverless Architectures*. [online].
Disponibile su <<https://martinfowler.com/articles/serverless.html>> [Data di accesso: marzo 2021].
 - *AWS Lambda - The Ultimate Guide*. [online].
Disponibile su <<https://www.serverless.com/aws-lambda>> [Data di accesso: marzo 2021].