SAPIENZA
UNIVERSITÀ DI ROMA

Progetto e Realizzazione di un broker per il calcolo ed il confronto dei costi di servizi cloud

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Informatica

Candidato

Spataro Lorenzo
Matricola 1946590

Relatore

Prof. Emiliano Casalicchio

Anno Accademico 2022/2023

Progetto e Realizzazione di un broker per il calcolo ed il confronto dei costi di servizi cloud

Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: spataro.1946590@studenti.uniroma1.it

Indice

1 Introduzione **1**
1.1 Motivazioni . . . 1
1.2 Obiettivi . . . 1
1.3 Struttura della relazione . . . 1

2 Il cloud computing e il cloud broker **3**
2.1 Il cloud computing . . . 3
2.1.1 Le parti coinvolte . . . 3
2.1.2 Caratteristiche essenziali . . . 4
2.1.3 Modelli di servizio . . . 4
2.1.4 Modelli di distribuzione . . . 6
2.2 Cloud service broker . . . 6
2.2.1 Servizi offerti . . . 6
2.2.2 Normative giuridiche e qualità del servizio . . . 7
2.2.3 Valutazione dei costi . . . 8

3 Il Price Broker: progettazione e tecnologie utilizzate **9**
3.1 Idea progettuale . . . 9
3.2 Funzionalità . . . 9
3.3 Descrizione implementativa . . . 10
3.4 Tecnologie e protocolli utilizzati . . . 12
3.4.1 Linguaggi di programmazione . . . 12
3.4.2 Architetture e protocolli . . . 13
3.4.3 Modelli di predizione . . . 14

4 Analisi dell'uso e stima dei costi: implementazione **16**
4.1 Interazione con i provider . . . 16
4.1.1 Amazon Web Services . . . 16
4.1.2 Microsoft Azure . . . 17
4.1.3 Google Cloud . . . 21
4.2 Raccolta metriche di utilizzo . . . 22
4.2.1 Amazon Web Services . . . 22
4.2.2 Microsoft Azure . . . 28
4.2.3 Google Cloud . . . 32
4.3 Stima dei costi . . . 34
4.3.1 Amazon Web Services . . . 34

5 Previsione dell'uso: implementazione **39**
5.1 ARIMA . . . 40
5.2 DeepAR . . . 40

6 Test e conclusioni **44**
6.1 Test 44
6.1.1 Visualizzazione Istanze, raccolta metriche e calcolo dei costi . 45
6.1.2 Predizioni dell'uso 48
6.1.3 Conclusioni 49
6.1.4 Conclusioni Finali 50

Bibliografia **51**

Capitolo 1

Introduzione

1.1 Motivazioni

Al giorno d'oggi è frequente utilizzare servizi che permettono di archiviare immagini, analizzare dati ed effettuare varie operazioni computazionali senza possedere fisicamente l'hardware necessario. Il vantaggio principale di questi servizi, grazie alle tecnologie introdotte dal **Cloud Computing**, è quello di permettere la condivisione di risorse in modo da contenerne i costi che possono risultare onerosi per l'utente medio.

Data la grande espansione che sta avendo il Cloud Computing, può risultare complesso per un utente non esperto integrare i vari servizi cloud, soprattutto se appartengono a diversi provider, ma anche gestire l'uso delle risorse per migliorare le performance oppure semplicemente ridurre i costi qualora vi siano agenzie che offrano un piano cloud migliore a quello sottoscritto; a tal proposito potrebbe essere di grande aiuto la presenza di un'entità in grado di occuparsi della parte di stima e previsione dei costi in base all'utilizzo dei servizi: il **Price Broker**.

1.2 Obiettivi

L'attività di tirocinio ha avuto come scopo quello di creare uno strumento, in grado di osservare, prevedere l'utilizzo e stimare il costo, sostenuto da parte di un **Cloud Consumer** per quanto concerne i servizi di cloud offerti da tre dei principali Provider presenti sul mercato: *Amazon Web Services*, *Google Cloud*, *Microsoft Azure*. In particolare, il lavoro si è concentrato sulle quattro categorie in cui ricadono le risorse maggiormente utilizzate: **Macchine Virtuali**, **Account di archiviazione**, **Servizi di Containerizzazione e Database** e sulla base di queste è stato creato un applicativo che disponesse di tutte le funzionalità precedentemente accennate.

1.3 Struttura della relazione

Prima di procedere con la descrizione dettagliata del Price broker, è importante delinearne la struttura essendo la relazione suddivisa in quattro capitoli:

- **Il cloud computing e il cloud service broker**: introduzione alla tecnologia dietro ai servizi cloud.

- **Il Price Broker: progettazione e tecnologie utilizzate** illustrazione del tipo di applicazione che è stato creato, andando ad evidenziare i linguaggi di programmazione, i protocolli e i vari modelli utilizzati.
- **Analisi dell'utilizzo e stima del costo dei servizi: implementazione**: descrizione del processo per la raccolta delle caratteristiche tecniche, metriche di utilizzo in modo da ottenere il costa delle risorse adoperate.
- **Previsione dell'uso: implementazione**: analisi dettagliata della fase in cui si ipotizza l'uso futuro di un servizio nel futuro disponendo delle statistiche passate.
- **Conclusioni**: implementazione e aggregazione delle varie funzionalità esposte nei capitoli precedenti.

I due capitoli in cui verranno mostrate le implementazioni sono suddivisi in sottosezioni in base al provider di cui si parla.

Capitolo 2

Il cloud computing e il cloud broker

2.1 Il cloud computing

Secondo il **NIST** (National Institute of Standards and Technology), il Cloud computing è un modello che permette, attraverso la rete, di accedere comodamente e su richiesta ad un insieme di risorse di computazione configurabili come reti, server, spazi di archiviazione, applicazioni e servizi che possono essere fornite o liberate con non troppo sforzo di comunicazione dal provider.[3]

2.1.1 Le parti coinvolte

Le parti principali che comunicano per utilizzare, fornire e controllare le risorse sono le seguenti:

- **Cloud provider**: organizzazione responsabile di rendere disponibile i servizi cloud per le parti interessate, acquistando e gestendo l'infrastruttura su cui sono hostati ed eseguendo il software cloud per fornire il servizio attraverso Internet.
- **Cloud consumer**: persona od organizzazione che instaura un rapporto commerciale, per l'utilizzo dei servizi cloud, con un provider. Il consumer sceglie un servizio tra quelli offerti, si accorda per il prezzo ed infine esegue il servizio per il quale sarà richiesta un pagamento periodico con fatturazione annessa.
- **Cloud auditor**: è una parte terza che effettua una valutazione oggettiva ed indipendente dei servizi forniti da un cloud provider in termini di controlli di sicurezza, performance, impatto sulla privacy, ecc.
- **Cloud carrier**: agisce come intermediario per permettere la connettività e lo spostamento di dati tra consumer e provider attraverso internet e i dispositivi d'accesso (laptop, smartphone e computer).
- **Cloud broker**: è un'entità che gestisce l'uso, le performance e la distribuzioni dei servizi cloud, facendo da tramite per qualsiasi tipo di operazione tra cloud provider e consumer, la sua presenza non è necessaria.

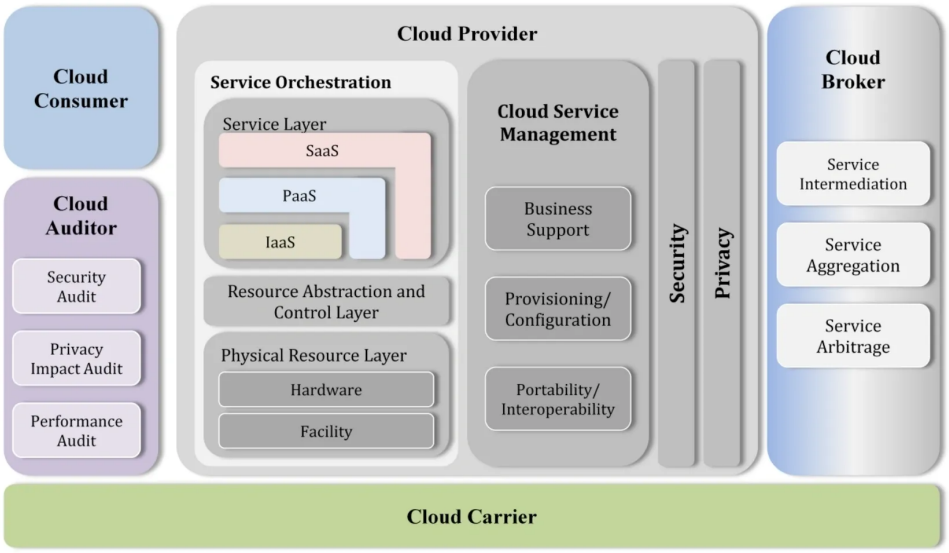

Figura 2.1. Architettura di riferimento del Cloud Computing secondo il NIST

2.1.2 Caratteristiche essenziali

Gli elementi chiave che delineano il panorama del cloud computing moderno sono:

- **On-demand self-service**: un utente può usare autonomamente le risorse che gli sono state assegnate, senza dover comunicare in modo diretto col provider.
- **Broad network access**: le varie funzionalità sono accessibili tramite Internet attraverso l'implementazione di meccanismi standard sui diversi tipi di dispositivi come laptop, smartphone e workstation.
- **Resource pooling**: il provider raggruppa le risorse computazionali in modo da servire più utenti utilizzando un modello che viene definito multi-tenant, col quale le risorse fisiche e virtuali vengono assegnate e rimosse in base alle esigenze. L'utente non conosce la posizione esatta delle risorse di cui usufruisce ma il provider potrebbe dargli la possibilità di scegliere tra varie località in cui si trovano i datacenter che ospitano i server.
- **Rapid elasticity**: capacità di aumentare o diminuire le risorse allocate per l'utente con modalità rapide tali che all'utente sembra di averle illimitate in qualsiasi momento.
- **Measured service**: ogni tipo di servizio viene sottoposto a controlli e ottimizzazioni utilizzando dei sistemi di misurazione delle risorse, le statistiche che vengono estratte sono disponibili sia al provider che all'utente.

2.1.3 Modelli di servizio

Esistono tre modelli principali che consentono ai consumer di sfruttare al meglio le risorse e la flessibilità della tecnologia cloud e sono:

- **Software as a Service** (SaaS): il provider consente all'utente di utilizzare le sue applicazioni in esecuzione su un infrastruttura cloud.[1]
  Le applicazioni sono accessibili attraverso i web client con semplice interfacce utente oppure interfacce di programmazione.
  L'utente non ha la possibilità di controllare l'infrastruttura sottostante come i vari server, il sistema operativo o la rete e può accedere solo ad un limitato numero di impostazioni relative all'applicazione.

- **Platform as a Service** (PaaS): il cliente del servizio ha la possibilità di distribuire, sull'infrastruttura cloud, applicazioni da lui create o semplicemente possedute e che siano compatibili con i linguaggi di programmazione, protocolli e servizi supportati dal provider. In questo caso l'utente ha il controllo sulle applicazione distribuite e sulle configurazioni dell'ambiente su cui sono hostate ma non può gestire l'infrastruttura.

- **Infrastructure as a Service** (IaaS): le risorse di computazione, archiviazione o di rete vengono messe a disposizione del consumatore grazie alle quali può distribuire ed eseguire qualsiasi tipo di software, incluso un sistema operativo. Con questo modello, l'utente ha il controllo sui sistemi operativi, l'archiviazione e le applicazioni distribuite e una serie di impostazioni per alcuni componenti di rete come firewall.

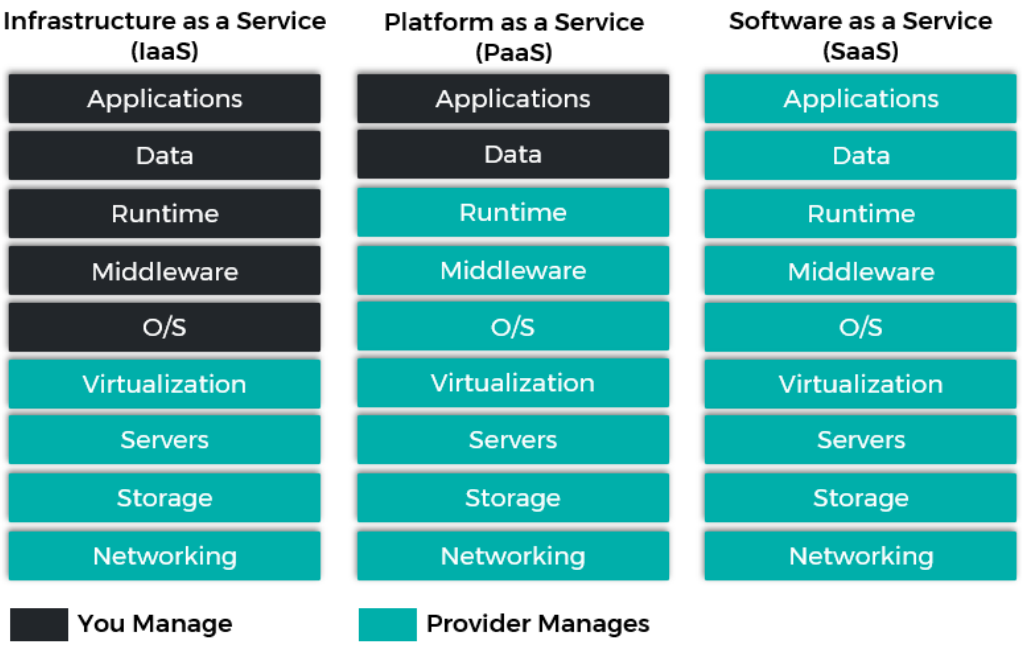

Figura 2.2. Gestione delle risorse

[1] L'insieme di hardware e software che rispetta le cinque caratteristiche principali del Cloud computing.

2.1.4 Modelli di distribuzione

I principali modelli di distribuzione che delineano come le risorse e i servizi vengono erogati e accessibili agli utenti vengono descritti di seguito:

- **Private cloud**: il provider rende disponibile un'infrastruttura all'uso esclusivo di una singola organizzazione che comprende vari utenti come quelle aziendali o universitarie. L'infrastruttura può essere di proprietà e gestita dall'organizzazione stessa o da una terza parte, può essere di due tipi: *on-premise*[2] oppure *off-premise*[3].
- **Community cloud**: un insieme di utenti appartenenti a diverse associazioni che possiedono gli stessi interessi (politici o di sicurezza) e che condividono l'uso di un'infrastruttura cloud gestita da una o più delle organizzazioni della community oppure da terze parti e anche in questo caso può essere hostata in locale da una delle organizzazioni oppure su dei server remoti.
- **Public cloud**: l'infrastruttura è aperta all'utilizzo di un pubblico qualsiasi, essa può essere sotto il controllo di un'azienda, università oppure di un'organizzazione governativa ed è localizzata nei datacenter del provider.
- **Hybrid cloud**: questo modello rappresenta l'unione di due o più infrastrutture che possono essere private, community o public, sono legate attraverso tecnologie standard che consentono la portabilità dei dati e delle applicazioni.

2.2 Cloud service broker

Come abbiamo visto, il cloud computing prevede diverse dinamiche che, a causa dell'espansione che avuto negli ultimi tempi, possono risultare complesse e impegnative al cliente.
Per ridurre le pratiche da sbrigare, il consumer potrebbe necessitare l'aiuto di un **Cloud Service Broker** (CSB), un entità in grado di gestire l'uso, le performance e la distribuzione di servizi cloud contattando al suo posto il provider effettuando gli accordi commerciali necessari.[2]

2.2.1 Servizi offerti

In particolare, un cloud service broker può fornire i seguenti tre servizi:

- **Service intermediation**: il broker migliora un determinato servizio andando a fornire dei servizi a valore aggiunto, ovvero non di base, all'utente. Ad esempio può occuparsi di riportare le performance, migliorare la sicurezza, gestire l'accesso ai servizi cloud ecc...
- **Service aggregation**: servizio che permette di combinare un insieme di servizi in uno o più nuovi in grado di comunicare tra loro garantendo l'integrazione e lo spostamento sicuro dei dati tra consumer e i provider. Ad esempio effettuando la portabilità di un'applicazione da un provider all'altro.

[2] Infrastruttura posseduta, gestita e resa operativa dall'organizzazione.
[3] Infrastruttura fornita e gestita da una terza parte

- **Service arbitrage**: indica la flessibilità da parte del broker di scegliere di affidarsi a diversi provider per i servizi che l'utente vuole utilizzare. In base a qualche criterio, come ad esempio il miglior rapporto qualità/prezzo analizzando le varie proposte in una classifica.

Figura 2.3. Scenario tipico con cloud broker

2.2.2 Normative giuridiche e qualità del servizio

Nell'era del cloud computing, un cliente si affida ad un provider per l'elaborazione e l'archiviazione dei dati e si potrebbero verificare casi in cui quest'operazione potrebbe risultare non conforme alla legislazione del cliente visto che, in generale nell'industria ICT, gli attori principali come provider, sviluppatori e clienti devono rispettare le leggi e i regolamenti che introducono restrizioni funzionali e non da tenere in considerazione in fase di progettazione dei sistemi ed in corso d'opera.

A tal proposito, se vi è un broker a svolgere il ruolo di intermediario tra provider e consumer, esso si dovrebbe occupare anche di gestire la parte legale e di regolazione degli accordi per conto delle due parti, andando ad esempio a monitorare i cambiamenti legali, le metriche di livello del servizio, verificare che i servizi siano conformi agli SLA e alla legge durante tutte le fasi dei vari servizi cloud (run-time, aggregazione, composizione, ottimizzazione e direzione).

Secondo una ricerca [1], quattro delle funzionalità più richieste per un broker, affinchè sia consapevole delle norme giuridiche vigenti, sono le seguenti:

- **Legal-rule compliance checking**: indica la capacità del broker di controllare in ogni momento che il provider e i servizi cloud siano conformi ad una normativa, che sia sovrannazionale o nazionale.
- **Legislation dynamic management**: il broker deve essere in grado di tracciare dinamicamente i cambiamenti legislativi o delle caratteristiche dei servizi che potrebbero portare ad una violazione legale dei requisiti.
- **QoS monitoring**: il broker deve monitorare e analizzare le metriche di qualità del servizio sia per controllare la conformità legale che il rispettto degli SLA. Questa funzionalità deve essere scalabile affinchè riesca a gestire la quantità di dati proveniente da tutte le istanze di un servizio.
- **Seamless service migration**: abilità nel definire ed effettuare un piano di migrazione di servizi cloud, riducendo il tempo di inattività grazie all'uso di formati standard per i dati ed ambienti indipendenti dalla piattaforma.

2.2.3 Valutazione dei costi

Grazie alla funzionalità del service arbitrage, un cloud service broker è in grado di integrare alcuni servizi e fornire i permessi per selezionarli da dei provider in base a dei criteri. Secondo vari studi, uno dei più importanti parametri nel selezionamento di un provider adatto per l'utente è il costo.[4] La maggior parte dei broker, permette di pagare lo stesso costo per gli stessi servizi ma per considerare un broker più efficace di un altro, bisogna analizzare il meccanismo che utilizzano per ottimizzare il rapporto qualità prezzo dei servizi.
Le attività che un CSB ideale dovrebbe offrire per raggiungere un buon livello di efficienza sono :

- **Analisi delle esigenze del consumer**: Un CSB inizia con un'analisi dettagliata delle esigenze dell'utente. Questo può includere la valutazione delle risorse necessarie, delle prestazioni richieste e dei requisiti di sicurezza.
- **Confronto tra provider**: i CSB valutano i vari fornitori di servizi cloud disponibili per trovare quelli che meglio soddisfano le esigenze dell'utente. Questa valutazione comprende non solo i costi diretti, ma anche altri fattori come la reputazione del fornitore, la qualità del servizio, la conformità normativa, ecc.
- **Ottimizzazione dei Costi**: una volta identificati i fornitori di servizi cloud appropriati, il CSB cerca di ottimizzare i costi. Questo può comportare la selezione di offerte personalizzate, la negoziazione di tariffe speciali o la raccomandazione di strategie di utilizzo delle risorse per massimizzare l'efficienza e ridurre i costi.
- **Monitoraggio costante**: i CSB non limitano la loro attività alla fase iniziale. Mantengono un monitoraggio costante dei costi e delle prestazioni per assicurarsi che gli utenti stiano ottenendo il massimo valore dai servizi cloud.

Capitolo 3

Il Price Broker: progettazione e tecnologie utilizzate

3.1 Idea progettuale

Per far fronte al problema di ottimizzazione del costo, è stato progettato uno strumento che permetta di arricchire le funzionalità del Cloud Service Broker nell'ambito prettamente economico e statistico dei servizi: il **Price Broker**.
E' stato ideato al fine di poter fornire un servizio a valore aggiunto di valutazione e comparazione dei costi di implementazione e di monitoraggio dei costi operativi.
I costi possono riferirsi sia al listino prezzi standard del provider oppure ad un accordo particolare e specifico per quell'organizzazione.

3.2 Funzionalità

Nel dettaglio, i servizi che mette a disposizione il Price broker sono principalmente i seguenti tre:

- **Service usage monitor**: i costi di un servizio cloud sono definiti in base all'utilizzo che se ne fa ed in particolare vi sono delle metriche, in base al tipo di risorsa, come il tempo di utilizzo, il traffico di rete, il throughput. Ad esempio le istanze delle macchine virtuali hanno un costo che va in base all'utilizzo della CPU come anche i database oltre al numero di *IOPS*[1], gli account di archiviazione alla quantità di byte immagazzinati e al numero di richieste che vengono fatte al bucket ecc. Il service usage monitor ha l'intento di raccogliere le varie statistiche sull'utilizzo di ogni servizio, quindi deve interagire con i vari provider che sono gli erogatori delle metriche ed utilizzare delle strutture standard ben definite per fornirle ai servizi successivi che si occuperanno di processarle.
- **Cost evaluation**: questa componente apprende prima il modo in cui ogni provider determina il costo finale di ogni servizio nella fattura, quindi quale metrica viene utilizzata, come è effettuato il calcolo e così via. Successivamente viene a conoscenza dal provider del prezzo che ha quella risorsa ed infine, facendo affidamento su quanto collezionato dal service usage monitor, valuta il costo finale di un servizio.

[1] Operazioni Input/Output

- **Cost prediction**: anche questa componente si affida ai dati forniti dal monitor ma in questo caso per effettuare una predizione dell'utilizzo futuro di un servizio in maniera tale da ottenerne automaticamente il prezzo in base al modello di costo del provider.

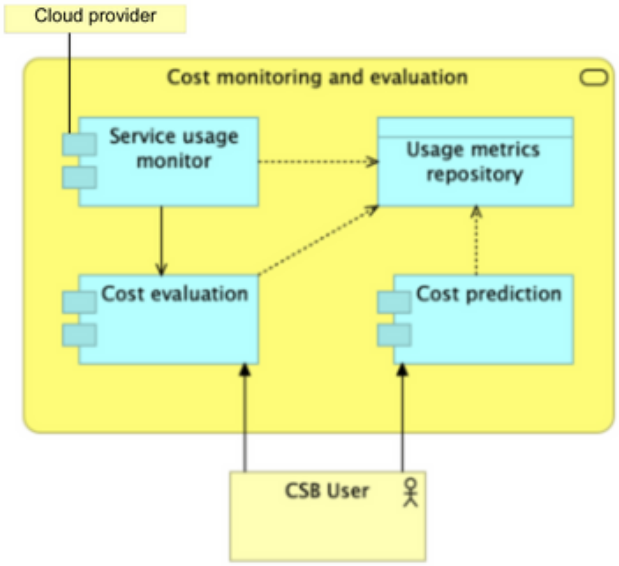

Figura 3.1. Architettura del Price Broker

3.3 Descrizione implementativa

I risultati delle varie funzionalità offerte vengono richiesti e mostrati attraverso l'utilizzo di un'applicazione web che dispone una semplice interfaccia utente. Il prerequisito fondamentale è stato di permettere ai service broker, che vogliano integrare questa componente [4] di poter mediare tra provider e qualsiasi tipo di cliente (anche il meno esperto) richiedendo il minor numero di informazioni sui servizi richiesti. In tal senso infatti, per poter effettuare la valutazione e la predizione dei costi dei occorre solamente possedere un account di accesso alla console di gestione dei servizi di un provider, in maniera tale che il provider possa riconoscere l'utente e far sì che il broker possa raccogliere le statistiche sull'utilizzo necessarie. Scendendo nel dettaglio dell'applicazione, con l'interfaccia è possibile scegliere il provider con il quale mettersi in contatto, successivamente ci sarà un'altra schermata in cui si può optare per una categoria di servizi cloud. All'interno di ogni categoria vengono visualizzate le istanze create dall'utente con varie informazioni ad esempio lo stato (in esecuzione, in attesa, stoppata), il tipo di istanza, software utilizzati e altre caratteristiche secondarie dipendenti dal servizio cloud. Ogni istanza è infine selezionabile in modo da poterne raccogliere le metriche di utilizzo ed effettuare dunque le analisi previste sui costi.

Lo sviluppo dell'applicazione si è concentrato sul raccogliere le informazioni sull'utilizzo e lista dei prezzi delle seguenti quattro categorie di servizi cloud:

- **Macchine virtuali**: sono versioni digitali di un computer fisico. Il software delle macchine virtuali è in grado di eseguire programmi e sistemi operativi, archiviare i dati, connettersi alle reti e svolgere altre funzioni di calcolo, e richiede operazioni di manutenzione quali aggiornamenti e monitoraggio del sistema proprio come qualsiasi computer. Su un computer possono essere installate diverse macchine virtuale attraverso un processo chiamato *virtualization* che consente l'astrazione delle componenti hardware andando ad utilizzare le risorse efficientemente e poter assegnarle ad ogni macchina in forma di risorsa virtuale.

- **Servizi di archiviazione**: permettono l'archiviazione di dati e file e l'accesso tramite Internet o reti private dedicate. Il provider archivia, gestisce e mantiene in modo sicuro i server di archiviazione, l'infrastruttura e la rete per garantire l'accesso ai dati quando ne hai bisogno su scala praticamente illimitata e con capacità elastica. Lo storage nel cloud elimina la necessità di acquistare e gestire la propria infrastruttura di storage dei dati, garantendo agilità, scalabilità e durabilità, con accesso ai dati in qualsiasi momento e ovunque. I file vengono raccolti in dei contenitori virtuali chiamati **bucket**, offrendo così un modo organizzato e scalabile per archiviare e gestire dati.

- **Servizi di containerizzazione**: permettono di distribuire il software raggruppando il codice di un'applicazione con tutti i file e le librerie di cui ha bisogno in un singolo eseguibile leggero, chiamato container, il quale viene astratto dal sistema operativo dell'host e, quindi, è autonomo e diventa portatile, in grado di funzionare su qualsiasi piattaforma o cloud, senza problemi. A differenza della virtualizzazione che mira a far girare diversi sistemi operativi su un singolo server, la containerizzazione esegue un singolo sistema operativo con diversi spazi che isolano i processi tra loro. Il software che permette ai container di gestire i container, funzionando come da sistema operativo e virtualizzando le risorse di un server come le macchine virtuali, si chiama **Docker**.

- **Database**: è molto simile a un database gestito in loco, tranne che per la gestione dell'infrastruttura che è rappresentata da delle macchine virtuali utilizzate e dedicate ad archiviare dati; il database viene supervisionato e gestito dal consumer.

L'applicazione inoltre, allo stato attuale, prevede il supporto dei tre provider leader del cloud computing con alcune caratteristiche differenziate:

- **AWS (Amazon Web Services)**: accesso con identità e chiave segreta che vanno creati dalla console di gestione dei servizi. Seguendo lo stesso ordine delle categorie, i servizi supportati dalle funzionalità del price broker sono:

  - **Amazon Elastic Compute Cloud** (EC2)
  - **Amazon Simple Storage Service** (S3)
  - **Amazon Elastic Container Service** (ECS)
  - **Amazon Relational Database Service** (RDS)

- **Microsoft Azure**: accesso con email e password dell'account. Azure come AWS supporta tutte le categorie con i seguenti servizi

  - **Azure Compute**

  - **Azure Storage Account**
  - **Azure Container Istances**
  - **Azure SQL Database**

- **Google Cloud**: come azure, si possono richiedere le metriche e i prezzi con email e password dell'account ma questo provider invece, allo stato attuale, supporta solamente l'analisi dei costi per le macchine virtuali (**Compute Engine**) e gli account di archiviazione (**Cloud Storage**)

3.4 Tecnologie e protocolli utilizzati

Poiché i requisiti del progetto richiedevano di effettuare delle valutazioni su dei dati raccolti e non c'era la necessità di archiviarli, si è deciso di creare il Price Broker sotto forma di *applicazione web* che collezionasse localmente al browser e alla sessione i dati effettuando in seguito le dovute analisi. Le cause che inoltre hanno fatto sì che si adottasse questa soluzione, sono dovute dal supporto che hanno le tecnologie web standard, su cui è basata l'app e che le permettono di essere eseguita su qualsiasi dispositivo con un browser moderno ma anche dalla semplicità di manutenzione e di distribuzione della stessa.

Per la parte della predizione invece, oltre all'interfaccia per consentire all'utente di eseguire i vari comandi e visualizzare le informazioni, è stata esposta un **API (Application Programming Interface)** [2]. In pratica, l'API consente di recuperare il risultato delle previsioni sui costi futuri semplificando la comunicazione con la macchina e consentendo agli utenti di ottenere facilmente e in modo efficiente le informazioni di previsione desiderate. Queste previsioni vengono generate mediante l'esecuzione di funzioni specializzate su una macchina dedicata, garantendo un'analisi accurata e tempestiva dei dati.

3.4.1 Linguaggi di programmazione

Al fine di snellire il codice, al posto di JavaScript puro (senza uso di librerie), l'applicazione è stata scritta facendo uso di un **framework**[3] open-source basato su una libreria (**React**) e denominato **Next.js**. Il motivo della scelta effettuata dipende dai seguenti servizi principali che offre il framework:

- **Scomposizione in componenti**: possibilità di suddividere l'interfaccia utente in piccoli pezzi riutilizzabili, facilitando la gestione del codice, rendendolo più modulare e manutenibile
- **Virtual DOM**: rappresentazione virtuale leggera del *DOM (Document Object Model)*[4] effettivo che permette di ottimizzare le operazioni di aggiornamento rendendo l'applicazione più efficiente in termini di prestazioni.
- **JSX**: estensione che consente di scrivere il markup dell'interfaccia utente direttamente nel codice JavaScript.

[2] Rappresenta un insieme di definizioni e protocolli con i quali vengono realizzati e integrati i software applicativi

[3] Strumento che permette di semplificare e velocizzare il processo di sviluppo quando si utilizza un linguaggio di programmazione

[4] Struttura dati che contiene la pagina e tutti i suoi elementi, come pulsanti, etichette e caselle a discesa

- **Routing dinamico**: semplifica la gestione del routing, consentendo la definizione di percorsi dinamici per le pagine. Ciò facilita la creazione di applicazioni a pagine singole (SPA) e di routing più complessi.

Per quanto concerne l'API e l'implementazione dei modelli di predizione, la scelta del linguaggio di programmazione è ricaduta su **Python** poichè possiede grande supporto ed è ampiamente utilizzato per applicazioni Web, nella data science e nell'analisi dei dati oltre ad essere efficiente, facile da imparare e con la possibilità di essere eseguito su diverse piattaforme. In particolare, è stato fatto uso di un framework Python chiamato **Django** che è possibile utilizzare per sviluppare applicazioni Web rapidamente. Django è una scelta appropriata poiché semplifica la gestione di funzionalità comuni nelle applicazioni Web, come l'autenticazione e il recupero di informazioni. Grazie alla sua architettura modulare, Django raggruppa queste funzioni in una vasta raccolta di moduli riutilizzabili, riducendo la complessità dello sviluppo.

3.4.2 Architetture e protocolli

L'architettura che caratterizza l'API delle predizioni è costituita da un client che effettua delle richieste (la web app con interfaccia) ed un server sempre in ascolto (la macchina che ospita l'API). Le due entità per comunicare e scambiarsi informazioni utilizzano il protocollo applicativo **HTTP (Hypertext Transfer Protocol)**.

L'API che riceve le richieste è stata creata con il toolkit *Django Rest Framework*, in maniera da renderla una struttura **REStful (Representational State Transfer)**. Una API RESTful è un'interfaccia che segue i seguenti principi architetturali:

- **Rappresentazione delle risorse**: le risorse, come dati o servizi, sono identificate da URL e rappresentate tramite dei formati standard di rappresentazione del dato.Sia le informazioni che il client invia all'API che le risposte rappresentate dai dati futuri sull'uso di un servizio cloud, vengono scambiate sottoforma di **JSON (JavaScript Object Notation)**.

```
{
  "menu": {
      "id": "file",
      "value": "File",
      "type": "text",
      "meta": []
  }
}
```

Esempio oggetto JSON

- **Stateless communication:** ogni richiesta del client al server deve contenere tutte le informazioni necessarie per comprendere e soddisfare la richiesta. Non si tiene alcuno stato sul server tra le richieste.
- **Operazioni CRUD tramite metodi HTTP**: Le operazioni di Creazione, Lettura, Aggiornamento e Eliminazione (CRUD) sono eseguite attraverso i metodi standard HTTP (POST, GET, PUT/PATCH, DELETE). Django Rest Framework semplifica l'implementazione di queste operazioni, consentendo di definire facilmente le azioni associate a ciascun metodo.

Come sarà possibile notare nella sezione 4.1, anche quest'ultimi espongono delle API con metodi standard per fornire le informazioni sui servizi cloud che offrono.

3.4.3 Modelli di predizione

Per effettuare le previsioni sull'utilizzo del servizio cloud, è stato adottato un approccio basato su due diversi modelli: l'**ARIMA(AutoRegressive Integrated Moving Average)** che è un modello statistico che utilizza le serie temporali per comprendere meglio un insieme di dati oppure per predire trend futuri e il **DeepAR**, un modello appartenente al campo del Machine Learning di previsione temporale utilizzato per affrontare il problema delle serie temporali, in particolare nella previsione di dati sequenziali futuri.

La scelta è ricaduta su questi due modelli poiché adatti ad analizzare le serie temporali che vengono fornite dai provider e che rappresentano l'utilizzo delle risorse di un servizio.

Vediamoli ciascun algoritmo di previsione nel dettaglio:

ARIMA

Il modello ARIMA è una tecnica di previsione robusta e ampiamente utilizzata che si basa su una combinazione di tre componenti principali:

- **Componente Auto-Regressiva (AR)**: Rappresenta la relazione tra l'osservazione corrente e le osservazioni precedenti nel tempo. Questo componente cattura i pattern di autocorrelazione nei dati.
- **Componente di Media Mobile (MA)**: Esprime la relazione tra l'osservazione corrente e l'errore residuo delle osservazioni passate. Questo componente aiuta a modellare gli errori casuali nella serie temporale.
- **Componente di Integrazione (I)**: Rappresenta il numero di differenziazioni necessarie per rendere la serie temporale stazionaria. La stazionarietà è importante poiché semplifica la modellazione dei pattern temporali.

L'allenamento del modello ARIMA coinvolge la scelta ottimale degli ordini ARIMA (p, d, q), dove "p" rappresenta l'ordine AR, "d" indica il grado di differenziazione, e "q" rappresenta l'ordine MA. L'allenamento è svolto sulle osservazioni storiche dell'utilizzo del servizio cloud, consentendo al modello di apprendere e adattarsi ai pattern temporali presenti nei dati.

L'ARIMA è particolarmente adatto per modellare serie temporali stazionarie, fornendo previsioni precise basate su tendenze passate e relazioni temporali intrinseche. La sua applicazione nel contesto dell'utilizzo del servizio cloud ci ha permesso di ottenere stime significative sulle variazioni future dell'uso delle risorse, contribuendo così a una gestione più efficace e ottimizzata dei servizi offerti.

DeepAR

Il **Machine Learning** (ML) rappresenta un approccio computazionale in cui i computer apprendono automaticamente da dati passati per eseguire compiti specifici senza essere esplicitamente programmati. Tra le varie categorie di apprendimento automatico, il **Deep Learning** è un campo più specifico che si basa su reti neurali profonde per modellare e apprendere rappresentazioni complesse dai dati.

Il DeepAR è un modello di Deep Learning che consente una previsione più flessibile e accurata dell'utilizzo del servizio cloud, tenendo conto delle dinamiche complesse presenti nei dati a differenza dell'ARIMA più idele per serie stazionarie e lineari.

Ecco alcune caratteristiche principali:

- **Architettura a rete neurale riccorente (RNN)**: tipo di rete neurale adatta per modellare sequenze temporali. Le reti neurali ricorrenti consentono al modello di catturare le dipendenze temporali nei dati, prendendo in considerazione le informazioni precedenti nella sequenza.
- **Adattamento automatico**: capacità di adattarsi automaticamente a diversi pattern temporali presenti nei dati senza la necessità di specificare manualmente *iperparametri*[5] complessi. Il modello è in grado di apprendere e adattarsi a diverse frequenze, trend e *stagionalità*[6] presenti nelle serie temporali.
- **Generazione di distribuzioni**: a differenza di molti modelli di previsione che forniscono solo stime puntuali, DeepAR genera distribuzioni di probabilità per ogni passo temporale futuro. Ciò significa che il modello può fornire intervalli di confidenza per le previsioni, consentendo una comprensione più approfondita della variabilità prevista.
- **Utilizzo di variabili ausiliarie**: supporto dell'uso di variabili ausiliarie, che possono essere utilizzate per migliorare la previsione incorporando informazioni aggiuntive che potrebbero influenzare la serie temporale.

L'implementazione del modello è stata possibile grazie all'uso della libreria **PyTorch Forecasting** che ha semplificato l'implementazione e l'addestramento di modelli di deep learning, fornendo al contempo flessibilità e prestazioni ottimali.

[5] Parametri il cui valore viene utilizzato per controllare il processo di apprendimento
[6] Presenza di variazioni che si verificano a intervalli regolari specifici inferiori a un anno

Capitolo 4

Analisi dell'uso e stima dei costi: implementazione

4.1 Interazione con i provider

Prima di procedere col comprendere quali dati vengono raccolti e analizzati per calcolare i costi di un servizio, è opportuno capire come avviene la comunicazione tra l'applicazione web che ospita l'interfaccia e il cloud provider al fine di ottenere le informazioni necessarie e il tipo di messaggi che vengono scambiati.

Come precedentemente accennato, ogni provider mette a disposizione delle API a cui effettuare richieste per acquisire le specifiche di un servizio, sapere quanto è stata utilizzata una sua risorsa, ricavare i prezzi di un piano cloud ecc.

Vediamo nel dettaglio cosa mette a disposizione ciascun provider.

4.1.1 Amazon Web Services

Per rendere più semplice e veloce la comunicazione con AWS, il quale espone delle API per gestire e monitorare le risorse, è stato impiegato l'**SDK(Software Development Kit)**[1] L'SDK di JavaScript per AWS fornisce dei metodi che permettono di comunicare col provider senza utilizzare richieste HTTP esplicite. Le componenti del SDK che sono state utilizzate sono le seguenti:

- **CloudWatchClient**: Per ottenere e monitorare metriche e log attraverso il servizio Amazon CloudWatch, che offre una visione dettagliata delle risorse e delle applicazioni in esecuzione su AWS.
- **PricingClient**: Per recuperare informazioni sui prezzi dei vari servizi AWS, consentendo una valutazione precisa dei costi associati alle risorse utilizzate.

Ognuno dei due client permette di comunicare con le API omonime esposte da AWS.

Si aggiungono poi altre quattro che svolgono sempre la funzione da client per accedere a ciascun dei quattro servizi cloud ed osservarne le specifiche hardware/software: **S3Client, EC2Client, RDSClient e ECSClient.**

I client dispongono di un metodo per inviare dei comandi, tramutati poi in richieste HTTP, attraverso le quali effettuare le vere e proprie operazioni sulle risorse.

[1] Insieme di strumenti specifici per una piattaforma, riuniti in un unico posto per poter sviluppare più rapidamente le applicazioni.

```
// Esempio di esecuzione di un comando
import { EC2Client } from "@aws-sdk/client-ec2";
const command = new CommandObject()
const response = await client.send(command);
```

Autenticazione

Ogni client di servizio deve essere inizializzato con le adeguate credenziali **IAM(Identity Access Management)**:

```
var S3 = require('aws-sdk/client/s3')

var s3 = new S3({
    apiVersion: '2006-03-01',
    region: 'us-west-1',
    credentials: {IAM_CREDENTIALS}
}
```

IAM è un servizio progettato per gestire l'accesso agli altri servizi AWS in modo sicuro e controllato, consente di creare e gestire identità e autorizzazioni per gli account AWS. Quando si utilizza un SDK (Software Development Kit) AWS, come l'SDK di JavaScript, per interagire con i servizi AWS tramite il codice, IAM svolge un ruolo fondamentale nel garantire che le operazioni siano autorizzate e che gli utenti o le applicazioni abbiano solo l'accesso necessario alle risorse. Il servizio si basa sui seguenti concetti chiave:

- **Utenti**: IAM consente di creare utenti con credenziali di accesso per accedere ai servizi AWS. Ogni utente può essere assegnato a gruppi e dotato di autorizzazioni specifiche.
- **Ruoli**: IAM consente di definire ruoli che possono essere temporaneamente assunti da utenti, servizi o risorse, garantendo l'accesso solo quando è necessario.
- **Policy**: Le politiche definiscono quali azioni sono consentite o negate su quali risorse. Possono essere associate a utenti, gruppi o ruoli.

Dunque le credenziali che vengono utilizzate devono appartenere ad un utente al quale sia associata una policy con le autorizzazioni alle operazioni sui sui quattro servizi.

4.1.2 Microsoft Azure

Per comunicare con il cloud provider di Microsoft invece, a causa di problemi di supporto con l'SDK di JavaScript che era la prima scelta per procedere con l'implementazione, è stato utilizzata la **Fetch API**[2]. Grazie a quest'interfaccia è possibile semplificare la gestione delle richieste e delle risposte HTTP, offrendo un modo moderno e flessibile per effettuare chiamate a servizi web. Azure espone diverse APIs in base al servizio:

- **Compute API**: Azure offre servizi di calcolo flessibili, consentendo la creazione e la gestione di risorse computazionali su richiesta.
- **Storage Service API**: è l'API del servizio di archiviazione di Azure che consente di caricare e scaricare oggetti.

[2]Interfaccia JavaScript per lo scambio di messaggi HTTP

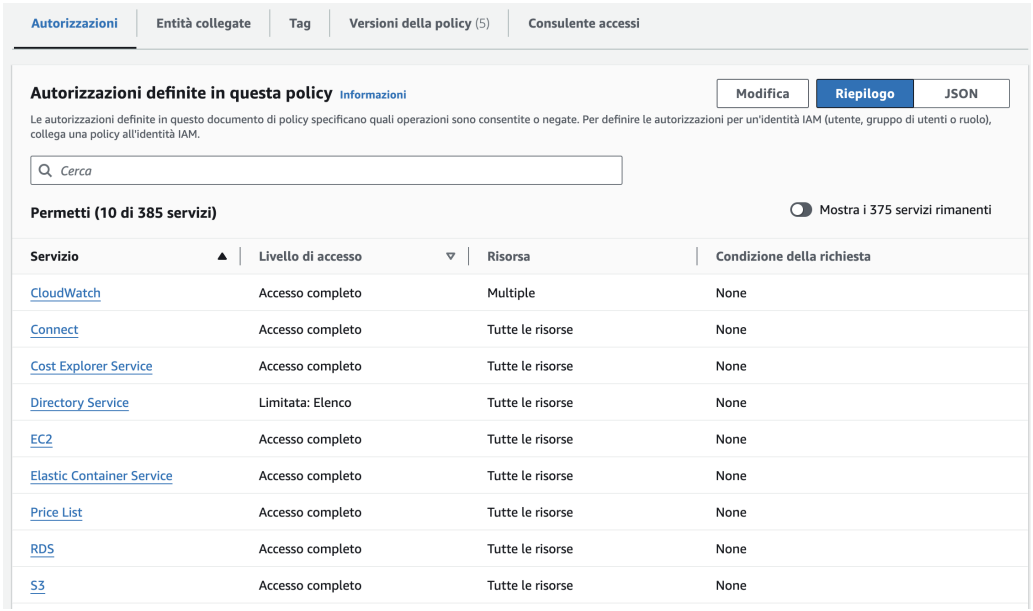

Figura 4.1. Visualizzazione dei permessi di una policy nella console AWS

- **SQL Database API**: consente agli sviluppatori di interagire con i database, ad esempio eseguendo query SQL e gestendo aspetti amministrativi.
- **Container Istances API**: consente di gestire e orchestrare le istanze di contenitori attraverso la creazione, avvio, arresto o la semplice visualizzazione.

Occorre sottolineare come le risorse di Azure siano gestite dal momento che per ogni account esiste un che contenitore logico che le raggruppa tutte e denominato **sottoscrizione**. Vi sono altri contenitori interni, chiamati **gruppi di risorse**, usati per raggruppare le risorse correlate in una sottoscrizione, di solito rappresentano una raccolta di asset necessari per supportare un carico di lavoro, un'applicazione o una funzione specifica all'interno di una sottoscrizione.

Autenticazione

Ogni richiesta che viene effettuata all'API di Azure deve contenere all'interno dell'header il token per autorizzare ed autenticare l'utente che fa la richiesta. Per ottenere un token, è necessario seguire il flusso **OAuth 2.0** o **OpenID Connect**. In Azure, la gestione dell'identità e l'emissione dei token sono gestite da **Azure Active Directory** (Azure AD). Ecco una spiegazione generale del processo:

- **Registrazione dell'applicazione in Azure AD**: prima di ottenere un token, è necessario registrare la tua applicazione, che utilizzerà le credenziali, in Azure AD. Questo processo ti fornirà l'*Application ID* (o Client ID) e il *Tenant ID* necessarie per le richieste di autenticazione.
- **Configurazione delle autorizzazioni**: vengono configurate gli URI di reindirizzamento successive alla concessione del token, il tipo di token da rilasciare e altre impostazioni avanzate.

- **Richiedere il token**: a questo punto inizia il flusso di autorizzazione OAuth 2.0. Active Directory offre la possibilità di utilizzarne diversi in base al tipo di applicazione.

Poiché l'app del Price Broker è di tipo *client-side*, è stato necessario utilizzare il **Flusso di Autorizzazione Implicito** che prevede i seguenti passaggi:

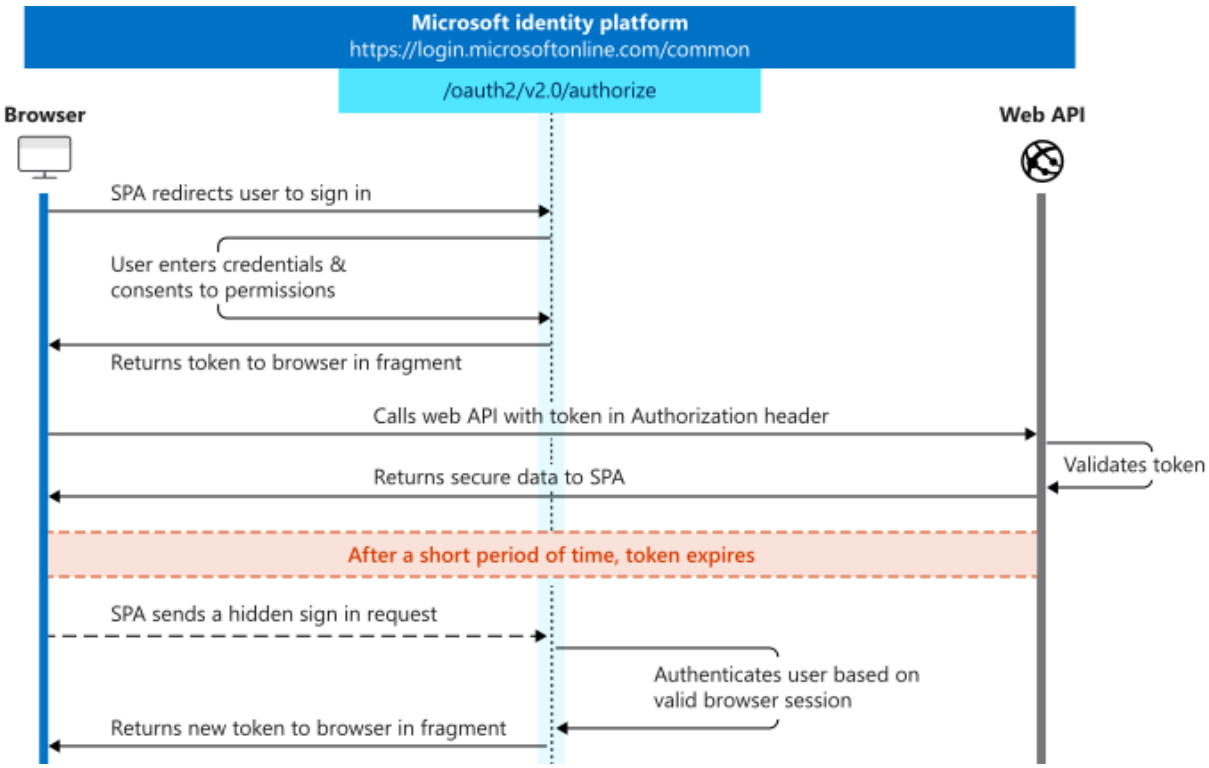

Figura 4.2. Flusso di Autorizzazione Implicito

1. Richiesta del token: per richiedere il token di accesso, vengono utilizzate, attraverso il passaggio per parametri nel *querystring*[3], le credenziali ricavate da Azure AD, lo *scope*[4] che verrà fatto delle credenziali quindi per quali app o risorse verrà utilizzato e il formato del token. Nel caso specifico del Price Broker viene utilizzato uno scope che garantisce l'impersonificazione del consumer dei servizi cloud.

2. Azure chiede all'utente il consenso: In questo passaggio, l'utente decide se concedere all'applicazione l'accesso richiesto. In questa fase, Azure mostra una finestra per il consenso che mostra il nome dell'applicazione e i servizi API di Azure a cui richiede l'autorizzazione per accedere con le credenziali di autorizzazione dell'utente, oltre a un riepilogo degli ambiti dell'accesso da concedere. L'utente può quindi acconsentire a concedere l'accesso a uno o più ambiti richiesti dalla tua applicazione o rifiutare la richiesta.

3. Il server OAuth 2.0 invia una risposta al redirectUri specificato nella richiesta del token di accesso. Se l'utente approva la richiesta, la risposta contiene un

[3]Parte dell'URL che contiene dei dati da utilizzare come parametri dal web server
[4]Tipo di utilizzo a livello applicativo

token di accesso. Se l'utente non approva la richiesta, la risposta contiene un messaggio di errore. Il token di accesso o il messaggio di errore viene restituito sul frammento hash dell'URI di reindirizzamento.

4. Scadenza del token: dopo un lasso di tempo non prolungato, il token scade e ne viene ricevuto un altro in base allo stato della sessione del browser. Potrebbe essere necessario farne nuovamente richiesta.

Implementazione del flusso

```
/*
    Procedura per la richiesta di un access token al server OAuth 2.0
*/
static getToken(redirectUri: string, clientId: string, tenant: string) {

    let fragmentString = location.hash.substring(1);

    // Controlllo della presenza di parametri all'interno del querystring.
    let params: any = [];
    let regex = /([^&=]+)=([^&]*)/g, m;

    while (m = regex.exec(fragmentString)) {
        params[decodeURIComponent(m[1])] = decodeURIComponent(m[2]);
    }

    // Se non e' gia' stato ricevuto l'access token lo richiedo
    if (!params['access_token']) {
        let oauth2Endpoint ='https://login.microsoftonline.com/{tenant}/oauth2/v2.0/
            authorize';

        // Viene creato un form fittizzio per inserire i dati nel querystring della
            richiesta GET
        let form = document.createElement('form');
        form.setAttribute('method', 'GET'); .
        form.setAttribute('action', oauth2Endpoint);

        // Parametri necessari alla richiesta del token
        let params = {
            'client_id': clientId,
            'redirect_uri': redirectUri,
            'scope': 'https://management.azure.com/user_impersonation'
            'response_type': 'token'
        };

        // Creazione dei campi del form con i valori dei parametri
        for (let p in params) {
            let input = document.createElement('input');
            input.setAttribute('type', 'hidden');
            input.setAttribute('name', p);
            input.setAttribute('value', params[p]);
            form.appendChild(input);
        }

        // Aggiunta del form al DOM ed invio.
        document.body.appendChild(form);
        form.submit();
    } else {
        // Se  il token e' nell'URL, restituiscilo
        return params['access_token']
    }
}
```

4.1.3 Google Cloud

Anche per interagire con le REST APIs di Google Cloud, in questo caso per assenza di supporto di un SDK per le applicazioni lato client, si è optato per l'utilizzo della Fetch API. Google Cloud mette a disposizione una vasta gamma di API che oltre a consentire di visualizzare e operare in modo avanzato con le risorse cloud che sono coinvolte nel progetto del Price Broker, permettono di monitorarle. Vediamo nel dettaglio con quali si interagisce e le loro caratteristiche:

- **Cloud Monitoring API**: è il servizio che permette ad uno sviluppatore di accedere alle statistiche di utilizzo delle risorse cloud tramite .
- **Cost Estimation API**: servizio che offre il calcolo dei costi utile ad aiutare gli utenti per stimare e monitorare i costi associati all'uso dei loro servizi cloud.
- **Compute Engine API**: fornisce un'interfaccia per interagire con le risorse di calcolo, in particolare le istanze di macchine virtuali (VMs). Con questa API, è possibile:
  - Creare, avviare, arrestare e eliminare istanze di macchine virtuali.
  - Configurare le caratteristiche delle VM, come la CPU, la memoria e le immagini del sistema operativo.
  - Monitorare lo stato delle VM e ottenere informazioni dettagliate sulle prestazioni.
  - Automatizzare la gestione delle risorse di calcolo su Google Cloud.
- **Cloud Storage JSON API**: consente l'interazione avanzata con il servizio di archiviazione oggetti di Google Cloud Storage. Alcune delle funzionalità offerte includono:
  - Caricamento e download di file in e da bucket di archiviazione.
  - Gestione delle autorizzazioni e delle politiche di accesso ai dati.
  - Ricerca e navigazione tra i contenitori di archiviazione.
  - Gestione dei metadati associati ai file e ai bucket.
  - Esecuzione di operazioni di trasferimento di dati e sincronizzazione.

Prima di continuare con la prossima sezione, è necessario spendere due parole sul modo in cui vengono organizzate le risorse di un account: i **progetti** Google Cloud. I progetti sono la base per creare, abilitare e utilizzare tutti i servizi Google Cloud, tra cui la gestione delle API, l'abilitazione della fatturazione, l'aggiunta e la rimozione di collaboratori e la gestione delle autorizzazioni per le risorse Google Cloud.

Autenticazione

Le due APIs di Google Cloud, così come avviene con quelle di Azure, per stabilire l'identità dell'applicazione che fa la richiesta e per garantire che solo le entità autorizzate possano scambiare informazioni con successo, utilizzano un flusso di autenticazione implicito per le applicazioni web lato client.

Prima di richiedere il token di accesso, ci sono delle azioni preliminari da eseguire all'interno della console del provider nella sezione **API e Servizi**:

- **Abilitazione delle API per il progetto**: dopo aver selezionato il progetto, all'interno della **Libreria API**, si cerca l'API da abilitare e una volta trovata occorre semplicemente attivarla dal tasto.
- **Creazione credenziali di autorizzazione**: nella pagina **Credenziali**, occorre recarsi su **ID client Oauth**, selezionare il tipo di applicazione "Applicazione web" e compilare il modulo che viene fornito. Le applicazioni che utilizzano JavaScript per effettuare richieste API di Google autorizzate devono specificare le origini JavaScript autorizzate. Le origini identificano i domini da cui l'applicazione può inviare richieste al server OAuth 2.0
- **Identificare gli ambiti (scope) di accesso**: Gli ambiti consentono all'applicazione di richiedere l'accesso solo alle risorse di cui ha bisogno, permettendo al contempo agli utenti di controllare la quantità di accesso che concederanno all'applicazione.

Implementazione del flusso

E' praticamente uguale a quella di Azure solamente con un diverso server a cui fare richiesta per il token, le credenziali relative ad un progetto ed altri ambiti d'accesso da dichiarare. Nello specifico, ai fini del progetto, è stato necessario utilizzare i seguenti ambiti per monitorare e accedere alle risorse:

```
https://www.googleapis.com/auth/monitoring
https://www.googleapis.com/auth/compute
https://www.googleapis.com/auth/cloud-platform
```

4.2 Raccolta metriche di utilizzo

La collezione delle statistiche di uso di una risorsa di un servizio cloud, come anticipato nel capitolo 3, avviene dopo aver appunto selezionato l'oggetto che appartiene alla categoria di servizi scelta ancora prima. Però prima di poter scegliere una specifica risorsa, è indispensabile ricavare la lista con gli identificatori univoci delle varie entità e alcune informazioni che serviranno per il successivo calcolo del costo. Per ogni provider, viene mostrato il procedimento che viene attuato ogni qualvolta si seleziona una categoria di servizi e si vuole visualizzare l'elenco delle risorse ed analizzarle singolarmente. E' importante sottolineare che le unità di campionamento che vengono richieste sono distanziate di un'ora l'una dall'altra.

4.2.1 Amazon Web Services

L'SDK, attraverso gli oggetti che vengono istanziati e svolgono la funzione di client HTTP tra l'applicazione e l'API specifica di AWS, fornisce dei comandi specifici da fornire al client che poi li incapsula e successivamente invia sotto forma di richiesta HTTP per effettuare le operazioni effettive sui dati.
Prima di procedere con la descrizione dettagliata dell'implementazione dei client di ogni servizio, occorre parlare del monitoraggio di una risorsa. Come già anticipato, il CloudWatch client è l'entità che si occupa di ricavare le informazioni sull'uso di una risorsa e il modo in cui ciò avviene è grazie all'esecuzione del comando **GetMetricStatisticsCommand** che prende i seguenti parametri come input sotto forma di oggetto JSON:

- **Namespace***: è un parametro che raggruppa le metriche correlate. Ad esempio ,"AWS/EC2" o "AWS/S3".
- **MetricName***: Il nome specifico della metrica che si desidera monitorare, come "CPUUtilization", "NetworkIn", "RequestCount", ecc.
- **Dimensions**: consentono di identificare in modo univoco la risorsa associata alla metrica. Ad esempio, per le metriche EC2, le dimensioni potrebbero essere "InstanceId".
- **StartTime***: istante d'inizio del periodo di monitoraggio. Deve essere nel formato ISO 8601 UTC (ad esempio, 2016-10-03T23:00:00Z). Specificato come oggetto Instant.endTime (obbligatorio):
- **EndTime***: istante di fine del periodo di monitoraggio. Anch'esso deve rispettare il formato ISO.
- **Period***: periodo di campionamento in secondi. Indica l'intervallo tra ogni **datapoint**[5] restituito. Ad esempio, se il periodo è 300 secondi, verrà restituito un datapoint ogni 5 minuti.
- **Statistics**[1]: statistiche richieste per la metrica. Possono includere:
  - "Average": Valore medio.
  - "Sum": Somma totale.
  - "Minimum": Valore minimo.
  - "Maximum": Valore massimo.
- **Unit**: unità di misura della metrica, come secondi, byte, percentuale ecc. Nel dettaglio, gli oggetti JavaScript che vengono utilizzati sono di seguito rappresentati:

L'implementazione di quest'ultimo comando e del ricavo delle specifiche tecniche di ogni risorsa è descritto nelle sottosezioni successive.

EC2

Tramite l'EC2Client, avviene l'interazione con l'API Amazon Elastic Cloud per poter creare e gestire istanze EC2. L'API, attraverso lo scambio di comandi, consente di creare, configurare, osservare e gestire le istanze EC2, fornendo un'ampia gamma di opzioni per personalizzare le risorse virtuali in base alle esigenze specifiche dell'applicazione.
Poiché l'obiettivo primario è quello di visualizzare tutte le istanze EC2 di un consumer, è stato fornito come input al client il comando **DescribeInstancesCommand** al quale è possibile allegare dei parametri nel caso in cui si vogliano visualizzare solo delle istanze specifiche. Dopo aver inviato il comando, nel caso in cui la risposta HTTP non contenga errori, verrà restituito il seguente oggetto JSON nel corpo della risposta:

*Campo obbligatorio
[5]Parte di informazione che descrive un'unità di osservazione

```
// Risultato del comando DescribeInstances
{
  Reservations: [
    {
      Groups: [
        {
          GroupName: "STRING_VALUE",
          GroupId: "STRING_VALUE",
        },
      ],
      // Lista delle istanze
      Instances: [
        // Istanza singola
        {
           AmiLaunchIndex: Number("int"),
           ImageId: "STRING_VALUE",
           InstanceId: "STRING_VALUE",
           InstanceType: "STRING_VALUE"
           NetworkInterfaces: [...]
           Status: "active" | "stopped" | ...
           // Altre informazioni tecniche sull'istanza
           ...
        }
      ]
    }
 ]
}
```

Il vettore **Instances** viene utilizzato per renderizzare a video l'elenco delle istanze mostrando il loro **InstanceId**, **InstanceType** e lo **Status**. Attraverso l'interazione con l'interfaccia visiva, si seleziona una specifica risorsa e si procede con la richiesta di monitoraggio. Per quanto riguarda il monitoraggio, la statistica fondamentale da valutare per stimare i costi di questo servizio è quella relativa all'utilizzo della CPU. Questa metrica è identificata dal valore "CPUUtilization" del parametro metricName. La rappresentazione della statistica viene richiesta come datapoint separati da intervalli di un'ora il cui valore caratterizzi la percentuale media di utilizzo. L'input che viene fornito al comando GetMetricStatistics, che viene eseguito per ottenere le statistiche, è dunque il seguente:

```
{
    namespace: "AWS/EC2",
    metricName: "CPUUtilization",
    dimensions: [
        {
            name: "InstanceId",
            value: "SELECTED_ISTANCE_ID"
        },
    ],
    period: 3600,
    startTime: "TIMESTAMP_ISO_8601",
    endTime: "TIMESTAMP_ISO_8601",
    statistics: [
        "Average",
    ],
    unit: "Percent"
}
```

Il risultato sarà costituito da un oggetto JSON che contiene un vettore di datapoint ciascuno costituito dal valore, timestamp e unità della metrica specificata. Ecco un esempio:

```
{
    Datapoints: [
```

```
        {
            Timestamp: "2016-08-08T22:52:00Z",
            Average: 0.0,
            Unit: "Percent"
        },
        {
            Timestamp: "2016-08-08T22:47:00Z",
            Average: 0.17,
            Unit: "Percent"
        },
        {
            Timestamp: "2016-08-08T22:42:00Z",
            Average: 0.16,
            Unit: "Percent"
        },
        {
            Timestamp: "2016-08-08T22:37:00Z",
            Average: 0.17,
            Unit: "Percent"
        }
    ],
    Label: "CPUUtilization"
}
```

RDS

Comunicare con l'API del servizio per la gestione dei database relazionali, significa essere in grado, attraverso dei metodi, di eseguire query SQL di lettura e scrittura sui i dati memorizzati all'interno istanze di database ma anche pianificare l'esecuzione di operazioni di backup e ripristino oltre alle operazioni base per creare, elencare, modificare, cancellare le istanze. Il comando utilizzato dal client collegato all'API per richiedere le informazioni su tutti i database istanziati è il **DescribeDBInstancesCommand** al quale si può chiedere di filtrare in base ad una serie di parametri come l'identificativo di un'istanza di DB. Il response JSON sarà del tipo:

```
{
    Marker: "STRING_VALUE",
    // Array delle istanze
    DBInstances:
    [
        // Instanza DB
        {
          DBInstanceIdentifier: "STRING_VALUE",
          DBInstanceClass: "STRING_VALUE",
          Engine: "STRING_VALUE",
          DBInstanceStatus: "STRING_VALUE",
         },
         ...
    ],
 };
```

Anche per quanto riguarda questo servizio, viene richiesto a CloudWatch di fornire la media di utilizzo della CPU in unità temporali distanti di un'ora e sotto forma di percentuale. L'unico parametro che varia è il namespace che diventa "AWS/RDS".

ECS

Grazie all'uso del client ECS è possibile creare, configurare e gestire task e servizi basati su container all'interno di un **cluster**[6] ECS. Un cluster è costituito un gruppo di risorse di calcolo configurate per eseguire e gestire container Docker. Le risorse possono essere istanze EC2 o nodi **Fargate**. Fargate un servizio serverless che consente di eseguire container senza la necessità di gestire le istanze sottostanti. Un task invece rappresenta un'istanza in esecuzione di un'immagine di container, mentre un servizio definisce come mantenere in esecuzione un certo numero di istanze di task. Con il commando **DescribeContainerInstancesCommand** si riesce ad ottenere la lista dei container all'interno di un cluster, rappresentati sottoforma di **Amazon Resource Name(ARN)**[7], avendo una risposta di questo tipo:

```
{
   containerInstances: [
      {
         containerInstanceArn: "string",
         ec2InstanceId: "string",
         pendingTasksCount: number,
         registeredAt: number,
         registeredResources: [
            {
               doubleValue: number,
               integerValue: number,
               longValue: number,
               name: "string",
               stringSetValue: [ "string" ],
               type: "string"
            }
         ],
         "runningTasksCount": number,
         "status": "string",
      }
   ],
}
```

Il passo successivo prevede la scelta dell'istanza di un container dal vettore *containerInstances*. In seguito, è possibile richiedere a CloudWatch le statistiche della metrica **instance_cpu_utilization** utilizzando l'id dell'istanza (*ec2InstanceId*) su cui è in esecuzione il container ottenendo come risultato la lista dei datapoint come precedentemente osservato.

S3

Utilizzando l'S3Client, è possibile interagire con i bucket associati all'account del consumer caricando, scaricando o eliminando un oggetto al loro interno. Si possono creare nuovi bucket, modificare o eliminare quelli esistenti e ovviamente riceverne l'elenco completo. Per riuscire ad ottenere quest'ultimo, è necessario eseguire il comando **ListBucketsCommand**, il quale output risulta avere una rappresentazione simile a quella seguente:

```
{
  Buckets: [
    {
      CreationDate: "2012-02-15T21:03:02.000Z",
      Name: "examplebucket"
    },
```

[6] Gruppo di sistemi nodo che serve a eseguire applicazioni containerizzate
[7] Identificazione univoca di una risorsa AWS

```
    {
      CreationDate: "2011-07-24T19:33:50.000Z",
      Name: "examplebucket2"
    },
    {
      CreationDate: "2010-12-17T00:56:49.000Z",
      Name: "examplebucket3"
    }
  ],
  Owner: {
    DisplayName: "own-display-name",
    ID: "examplee7a2f25102679df27bb0ae12b3f85be6f290b936c4393484be31"
  }
}
```

Successivamente, scegliendo uno dei bucket visualizzati, si può effettuare la ricerca delle metriche che interessano ai fini del progetto. In particolare, ciò che effettivamente interessa per il passaggio successivo è raccogliere informazioni su quanti byte sono stati occupati dagli oggetti all'interno di un bucket la cui metrica che lo descrive è denominata **BucketSizeBytes** e anche sul numero totale di richieste HTTP effettuate al bucket (**AllRequests**).

In questo caso quindi occorre eseguire il comando GetMetricStatistics due volte poiché l'input della richiesta delle statistiche coinvolge metriche diverse.

```
// Quantita' di byte occupati
{
    namespace: "AWS/S3",
    metricName: "CPUUtilization",
    dimensions: [
        {
            name: "BucketName",
            value: "SELECTED_BUCKET_NAME"
        },
    ],
    period: 3600,
    startTime: "TIMESTAMP_ISO_8601",
    endTime: "TIMESTAMP_ISO_8601",
    statistics: [
        "Average",
    ],
    unit: "Bytes"
}

// Totale di richieste HTTP ricevute
{
    namespace: "AWS/S3",
    metricName: "AllRequests",
    dimensions: [
        {
            name: "BucketName",
            value: "STRING_VALUE"
        },
    ],
    period: 3600,
    startTime: "TIMESTAMP_ISO_8601",
    endTime: "TIMESTAMP_ISO_8601",
    statistics: [
        "Sum",
    ],
    unit: "Count"
}
```

I datapoint risultanti nel primo caso quindi vengono restituiti come quantitativo medio di byte occupati dall'inizio fino a quell'istante di tempo e per conoscere quanti byte effettivamente sono stati utilizzati occorre prendere la media dell'ultima ora ricevuta. Mentre per la seconda richiesta si riceve il quantitativo di richieste ricevute in un ora e per conoscere effettivamente il totale assoluto occorre sommare i valori di tutti gli istanti di tempo.

4.2.2 Microsoft Azure

L'individuazione delle risorse di ciascun servizio e la successiva raccolta delle metriche in questo caso può avvenire solamente tramite richieste HTTP effettuate con la Fetch API. Le richieste devono prevedere il seguente formato per accedere univocamente ad una risorsa:

```
/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>/providers/<
    provider>/<resource-name>/
```

- **subscription-id** : identificativo di una sottoscrizione.
- **resource-group-name**: rappresenta il nome del gruppo di risorse in cui è contenuta la risorsa. Il nome deve essere univoco all'interno della sottoscrizione.
- **provider**: variabile di percorso che rappresenta il nome del fornitore di risorse. Ad esempio, potrebbe essere Microsoft.Compute per il servizio di calcolo.
- **resourcename**: rappresenta il nome specifico della risorsa all'interno del gruppo di risorse e del tipo di fornitore.

Per quanto concerne il servizio di monitoraggio, denominato **Microsoft Insights**, è importante spiegare in generale e per ogni servizio cloud le modalità per ottenere le statistiche di una metrica. All'url che si interroga, il quale cambia in base alla risorsa e al tipo di servizio, devono essere specificati l'intervallo di tempo, la granularità dei datapoint e il filtro per ottenere le statistiche solo sull'istanza interessata e per farlo si indicano i valori rispettivamente per i parametri **timespan**, **interval** e **$filter**. L'implementazione effettiva viene descritta di seguito per ogni servizio.

Compute

Per le risorse computazionali si fa affidamento al servizio API **Microsoft.Compute** di Azure per ricavare la lista delle istanze. Implementazione:

```
const url = "https://management.azure.com/subscriptions/{subscriptionId}/providers/
    Microsoft.Compute/virtualMachines?api_version=2023-07-01"

fetch(url, {
    headers: {
        Authorization: Bearer {token},
        Content-Type: application/json
    }
}).then(response => response.json())

// Http response della richiesta
{
    // Elenco delle macchine virtuali
    value: [
        // Singola istanza
        {
```

```
        properties: {
              vmId: "{vmId}",
              availabilitySet: {
                "id": "/subscriptions/{subscriptionId}/resourceGroups/{
                    resourceGroupName}/providers/Microsoft.Compute/availabilitySets
                    /{availabilitySetName}"
              },
              hardwareProfile: {
                vmSize: "Standard_A0",
                vmSizeProperties: {
                  vCPUsAvailable: 7,
                  vCPUsPerCore: 14
                }
              },
              // Altre proprieta'
              ...
          },
      },
      ...
  ]
}
```

Dall'elenco si sceglie un'istanza e si procede col collezionare i datapoint sulle statistiche comunicando con il servizio di monitoraggio. In questo caso ci interessa sapere quanto è stata utilizzata la CPU attraverso la metrica **Percentage CPU**.

```
const url = "https://management.azure.com/subscriptions/<subscription-id>/
    resourceGroups/<resource-group-name>/providers/Microsoft.Compute/
    virtualMachines/<resource-name>/providers/microsoft.insights/metrics
?metricnames=Percentage CPU
&timespan=<starttime/endtime>
&interval=PT1H
&aggregation=Average"

fetch(url, {
    headers: {
        Authorization: Bearer {token},
        Content-Type: application/json
    }
}).then(response => response.json())
```

Il risultato sarà simile al seguente, con i datapoint situati nella proprietà "data" del primo e unico elemento del vettore "timeseries":

```
{
  timespan: "<starttime>/<endtime>",
  value: [
    {
      id: "/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>/
          providers/Microsoft.Compute/virtualMachines/<resource-name>/providers/
          microsoft.insights/metrics",
      name: {
        "value": "Percentage CPU",
      },
      unit: "Percent",
      timeseries: [
        {
          // Vettore dei datapoint
          data: [
            // Datapoint singolo
            {
              timeStamp: "2023-09-19T02:00:00Z",
              total: 2
            },
            {
```

```
            timeStamp: "2023-09-19T02:01:00Z",
            total: 1
          },
          ...
        ]
      },
      ...
    ]
  }
 ],
 "namespace": "Microsoft.Compute/virtualMachines",
 "resourceregion": "{region}"
}
```

Storage Account

In questo caso l'API coinvolta nel fornire le informazioni al Price Broker su gli account di archiviazione è la **Microsoft.Storage**. Senza mostrare nuovamente il codice, basti solo sapere che l'endpoint che viene contattato è il seguente:

```
https://management.azure.com/subscriptions/{subscriptionId}/providers/Microsoft.C
lassicStorage/storageAccounts
?api_version=2023-07-01
```

Il risultato sarà costituito dagli account associati alla sottoiscrizione dell'utente, restituiti in un JSON simile a quello che segue.

```
{
  // Lista degli account
  value: [
    // Account singolo
    {
      id: "/subscriptions/{subscription-id}/resourceGroups/res2627/providers/
          Microsoft.Storage/storageAccounts/sto1125",
      kind: "Storage",
      location: "eastus",
      name: "sto1125",
      properties: {
         ...
        },
      sku: {
        name: "Standard_GRS",
        tier: "Standard"
      },
      type: "Microsoft.Storage/storageAccounts"
    },
 ]
}
```

Le metriche che interessa analizzare per queste risorse sono due: lo spazio occupato dagli oggetti nell'archivio e il numero di transazioni (richieste API) effettuate su un account individuate.
Il metricName di ognuna è, rispettivamente, **UsedCapacity** e **Transactions**.
L'url della richiesta HTTP diventa:

```
https://management.azure.com/subscriptions/<subscription-id>/resourceGroups/
resource-group-name>/providers/Microsoft.Storage/storageAccounts/<account-
name>/providers/microsoft.insights/metrics
?metricnames=UsedCapacity,Transactions
&timespan=<starTime/endTime>
&interval=PT1H
```

Un risultato tipo che si ottiene è il seguente:

```
{
    timespan: "<starttime>/<endtime>",
    // Indica che i datapoint sono distanti di un'ora
    interval: "PT1H",
    value: [
        {
            id: "SAME_AS_ENDPOINT_PATH",
            type: "Microsoft.Insights/metrics",
            name: {
                value: "UsedCapacity",
            },
            unit: "Bytes",
            timeseries: [
                {
                    metadatavalues: [],
                    data: [
                        {
                            timeStamp: "2023-11-07T21:51:00Z"
                            value: <INT_NUMBER>
                        },
                        ...
                    ]
                }
            ],
            errorCode: "Success"
        },
        {
            id: "SAME_AS_ENDPOINT_PATH",
            type: "Microsoft.Insights/metrics",
            name: {
                value: "Transactions",
            },
            unit: "Count",
            timeseries: [],
            errorCode: "Success"
        }
    ],
    namespace: "Microsoft.Storage/storageAccounts",
    resourceregion: "eastus"
}
```

Containers e SQL Databases

Poiché l'implementazione del ricavo dei dati di ogni istanza è pressoché lo stesso e siccome non è necessario aggiungere altri esempi come precedentemente fatto, in questa sezione vengono descritti brevemente i servizi API utilizzati per le due categorie in questione e le metriche utili al Price Broker.

Iniziando dai Container, l'API che viene esposta è denominata **ContainerInstance API** e grazie alle sue funzionalità, essa consente di ricevere i container di un gruppo già conosciuto da visualizzare seguendo la falsariga dei due servizi cloud precedenti. L'endpoint da chiamare è:

```
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{reso
urceGroupName}/providers/Microsoft.ContainerInstance/containerGroups/{containerGr
oupName}
?api_version=2023-05-01
```

All'interno del JSON di risposta si trova la lista di istanze di container con ciascuna proprietà. Per quanto riguarda le metriche, quelle interessanti riguardano l'utilizzo della CPU da parte del container che viene scelto per il monitoraggio ed è identificata dal nome **CPUUsage** (restituisce il conteggio di CPU virtuali utilizzate) e

l'allocazione della memoria RAM **MemoryUsage**. Come ulteriore argomento della richiesta, viene assegnata una stringa a **$filter** per indicare che si vuole monitorare un'istanza di container in particolare all'interno del gruppo di container:

```
// dim indica la dimensione della metrica, in questo caso il nome dell'istanza
$filter = "dim eq <ContainerName>"
```

Se invece si vogliono osservare le istanze di basi di dati SQL c'è da comunicare con il servizio **Microsoft.SQL** e chiedere la lista dei database. Per effettuare la richiesta, si deve essere a conoscenza del server logico sotto il quale sono raggruppati grazie al quale è possibile amministrare account di accesso, regole del firewall, regole di controllo, criteri di rilevamento delle minacce e gruppi di failover automatico. L'endpoint da chiamare in questo ambito è:

```
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resou
rceGroupName}/providers/Microsoft.Sql/servers/{serverName}/databases
?api_version=2021-11-01
```

La metrica su cui viene analizzata la base di dati che viene indicata, è basata sull'utilizzo della CPU, il servizio di Azure Monitoring permette di ricavarne i datapoint attraverso la percentuale di uso con il valore del metricName settato a **cpu percent** similmente a quanto avviene con le macchine virtuali.

4.2.3 Google Cloud

La stessa implementazione che è stata attuata per Microsoft Azure viene rispecchiata sul provider di servizi cloud di Google. Infatti anche per le istanze di Google Cloud viene utilizzata la Fetch API per le comunicazioni tra client e web server. Al fine di poter mettere mano alle risorse, occorre innanzitutto scegliere un progetto sul cui lavorare e sotto il quale sono raggruppate le risorse che interessano al Price Broker.

Compute Engine API

Scelti i vari parametri, è possibile richiedere l'elenco delle risorse computazionali contattando l'endpoint del servizio in questo modo:

```
url = "https://compute.googleapis.com/compute/v1/projects/{project}/aggregated/
    instances"

fetch(url, {
    headers: {
        Authorization: Bearer {token},
        Content-Type: application/json
    }
}).then(response => response.json())

// Contenuto del response, le risorse sono raggrupate per zona di distribuzione
{
  kind: string,
  id: string,
  items: {
    zone_1: {
        // Lista istanze
        "instances": [
            // Istanza singola
            {
              kind: string,
              id: string,
              creationTimestamp: string,
```

```
            name: string,
            machineType: string,
            status: enum,
            zone: string,
            networkInterfaces: [...]
            // Altre proprieta'
            ...
          }
      ],
  ...
}
```

Andando a scegliere quale istanza monitorare attraverso l'identificativo, si può richiede alla Cloud Monitoring API i valori della metrica interessata, ovvero il tempo di utilizzo della CPU. All'interno dell'URL di chiamata, occorre specificare sempre il progetto oltre alla metrica, all'identificativo della risorsa e all'intervallo di tempo interessato, di seguito la costruzione dell'URL e il conseguente response del server:

```
url = "https://monitoring.googleapis.com/v3/projects/${this.project}/timeSeries
?filter=metric.type = 'compute.googleapis.com/instance/cpu/usage_time' AND
metric.labels.instance_name = 'my-instance-name'
&interval.startTime={startTime}
&interval.endTime={endTime}

// Response con le statistiche
{
  // Contiene un solo elemento con la metrica sulla CPU
  timeSeries: [
    {
      metric: {
        object (Metric)
      },
      resource: {
        object (MonitoredResource)
      },
      metadata: {
        object (MonitoredResourceMetadata)
      },
      metricKind: enum (MetricKind),
      valueType: enum (ValueType),
      points: [
        // Singolo datapoint
        {
          interval: {
            object (TimeInterval)
          },
          value: {
            object (TypedValue)
          }
        }
      ],
      unit: string
    }
  ],
  nextPageToken: string,
  executionErrors: [
    {
      object (Status)
    }
  ],
  unit: string
}
```

Cloud Storage JSON API

Ciò che cambia rispetto al interagire col precedente servizio API, è che in questo ambito occorre ottenere la lista dei bucket in cui vengono creati, caricati, scaricati e cancellati gli oggetti. Per procedere è necessario interrogare via HTTP il seguente endpoint:

```
https://storage.googleapis.com/storage/v1/b?project={projectName}
```

La risposta del server di Google è costituita da un oggetto con le caratteristiche tecniche di tutti i bucket del progetto indicato e il loro identificativo univoco.
Per le metriche, come avviene per gli altri provider, anche in questo caso è necessario prelevare le statistiche su quante richieste HTTP sono state effettuate al bucket **storage.googleapis.com/api/request_count** e quanti byte sono occupati dagli oggetti **storage.googleapis.com/storage/total_bytes**. L'URL seguente racchiude la richiesta per entrambe le informazioni:

```
https://monitoring.googleapis.com/v3/projects/{project_name}
?filter=(metric.type='storage.googleapis.com/api/request_count' OR
metric.type='storage.googleapis.com/storage/total_bytes') AND
resource.labels.bucket_name={bucket_name}
&interval.startTime={startTime}
&interval.endTime={endTime}
```

Il server risponde con un oggetto contenente due serie temporali riguardo le informazioni richieste.

4.3 Stima dei costi

4.3.1 Amazon Web Services

Un modo conveniente e centralizzato per interrogare AWS per informazioni sui prezzi dei prodotti cloud è fornito dalla **AWS List API**. L'API in questione utilizza degli attributi come la zona di distribuzione, l'archiviazione, il sistema operativo ed un'altra serie di filtri per restituire i prodotti che si cercano. Allo scopo del progetto quindi è necessario utilizzarla per ricevere il prezzo delle istanze e calcolarne il successivo costo in base alle statistiche sull'uso ricavate. Il prezzo viene restituito, in base al servizio, sotto forma di tariffa unitaria (di solito 1 ora) da pagare, ad esempio le macchine virtuali hanno un costo che varia in base a quante ore vengono tenute in esecuzione. Per la parte d'implementazione, continua ad essere comodo l'SDK che fornisce un client (PricingClient) apposito per comunicare con il web server predisposto. Nel dettaglio viene recuperato il prezzo di ciascuno dei quattro servizi eseguendo il comando **GetProductsCommand** al quale è possibile passare in input il **ServiceCode** di un servizio cloud a cui appartengono i prodotti di cui si desidera ricevere le informazioni (es. 'AmazonEC2') e una lista di filtri ciascuno sotto forma di oggetto con le seguenti proprietà:

- **Field**: il campo del prodotto sul quale effettuare la corrispondenza. Ad esempio l'identificativo dell'istanza oppure la regione di distribuzione.
- **Type**: il filtro che si vuole applicare.
  L'unico valore valido è: **TERM_MATCH** che restituisce solo i prodotti che hanno una corrispondenza tra il Field a il Value fornito.
- **Value**: il valore assunto dal Field per il quale si vuole filtrare.

Un esempio di esecuzione del comando:

```
const client = new PricingClient(config);
const input = {
    Filters: [
        {
            Type: "TERM_MATCH",
            Field: "ServiceCode",
            Value: "AmazonEC2"
        },
        {
            Type: "TERM_MATCH",
            Field: "volumeType",
            Value: "Provisioned IOPS"
        }
    ],
    FormatVersion: "aws_v1",
    NextToken: null,
    MaxResults: 1,
    ServiceCode:"AmazonEC2"
}
const command = new GetProductsCommand(input);


// Response
{
    FormatVersion: "aws_v1",
    NextToken: "57r3UcqRjDujbzWfHF7Ciw==:ywSmZsD3mtpQmQLQ5XfOsIMkYybSj+vAT+kGmwMFq+
        K9DGmIoJkz7lunVeamiOPgthdWSO2a7YKojCO+zY4dJmuNl2QvbNhXs+
        AJ2Ufn7xGmJncNI2TsEuAsVCUfTAvAQNcwwamtk6XuZ4YdNnooV62FjkV3ZAn40d9+wAxV7+
        FImvhUHi/+f8afgZdGh2zPUlH8jlV9uUtj0oHp8+DhPUuHXh+WBII1E/aoKpPSm3c=",
    PriceList: [
        {
            product: {
                productFamily: "Compute"
                attributes: {..}
            },
            serviceCode: "AmazonEC2"
            terms: {
                OnDemand: {
                    unit: "Hours/Mo"
                    pricePerUnit: {
                        USD: 0.1380000000
                    },
                    sku: "..."
                }
            },
            ...
        }
    ]
}
```

Per ogni servizio vengono elencati i filtri i cui Field and Value vengono ottenuti dal passaggio precedente di raccolta delle informazioni tecniche e come viene calcolato il costo:

- **Elastic Compute Cloud**:
  - **serviceCode** = 'AmazonEC2'
  - **instanceType** = 'STRING_VALUE'

  Il prezzo è relativo ad un'ora di utilizzo di CPU. Il costo è calcolato ottenendo le ore di utilizzo della CPU dalla raccolta metriche e moltiplicandolo per il costo unitario.

- **Relational Database Service**:
  - **serviceCode** = 'AmazonRDS'
  - **instanceType** = 'STRING_VALUE'
  - **engine** = 'STRING_VALUE'

  Il costo viene calcolato allo stesso modo del servizio precedente.

- **Elastic Container Service**:
  - **serviceCode** = 'AmazonECS'
  - **operatingSystem** = 'STRING_VALUE'
  - **cpuArchitecture** = 'STRING_VALUE'

  Il calcolo dei costi per questo servizio è simile a quello dell'EC2 visto che occorre analizzare l'uso della cpu da parte delle istanze di container.

- **Simple Storage Service**:
  - **serviceCode** = 'AmazonS3'
  - **usageType** = 'Requests-Tier1'
  - **usageType** = 'TimedStorage-ByteHrs'

  Viene restituito il prezzo in base a quanti gigabyte vengono occupati in un mese e il costo di una richiesta HTTP. Il costo è calcolato in base a quante richieste HTTP vengono fatte al bucket e quanto spazio risulta occupato.

Microsoft Azure

L'approccio per Azure è leggermente diverso poichè non si ha il supporto di un SDK che semplifica il lavoro di composizione dell'URL e delle richieste. L'API di cui parliamo in questo ambito è la Cost Management che espone un endpoint

`https://prices.azure.com/api/retail/prices?api=?-=?version=2023-01-01-preview`

al quale richiedere, tramite filtraggio effettuato nel querystring la lista dei prezzi di un servizio. Le risorse che abbiamo ottenuto nell'implementazione che riguarda la raccolta delle metriche contengono lo SKU grazie al quale è possibile al prodotto e ottenerne il prezzo. La procedura prevede di fornire lo sku e il tipo di prezzo che si vuole ricevere in input al parametro **$filter** in questo modo:

```
filter = "armSkuName eq '{sku_name}' and priceType eq 'Consumption'"

const url = "https://prices.azure.com/api/retail/prices?api-version=2023-01-01-
    preview?$filter={filter}"
```

Il risultato risulta essere simile all'oggetto JSON rappresentato di seguito

```
{
    currencyCode: "USD",
    tierMinimumUnits: 0.0,
    retailPrice: 2.305,
    unitPrice: 2.305,
    armRegionName: "southindia",
    skuName: "M8ms",
    serviceName: "Virtual Machines",
    serviceFamily: "Compute",
    unitOfMeasure: "1 Hour",
    type: "Consumption",
},
```

Oltre all'identificativo del prodotto e al tipo di prezzo, è possibile richiedere un'altra valuta, filtrare la categoria di prodotto, la regione e altri parametri secondari

Google Cloud

Questo provider, con l'esposizione della Cost Estimation API, prevede che, venga inviato un payload contenente l'utilizzo tipo di ogni risorsa in termini di tempo e di quantità e le caratteristiche tecniche (o tipo standard) per ciascuna, in modo che venga ricevuto il costo per quello che viene chiamato **workload** (carico di lavoro) e anche per ciascuna risorsa in modo separato. A differenza degli altri due provider, il costo viene quindi direttamente calcolato dal server in base alle informazioni fornite. L'URL che si deve interrogare via richiesta HTTP di tipo POST e il formato dell'oggetto che si deve inviare è descritto nell'implementazione di seguito

```
const url = "https://cloudbilling.googleapis.com/v1beta:estimateCostScenario?key=
    API_KEY"

// Carico di lavoro stimato per 1 ora per cinque macchine virtuali con 4 vCPU, 4 GB
     di Ram
const requestBody = {
    costScenario: {
        scenarioConfig: {
          estimateDuration: "3600s" // 1h
        },
    }
    workloads: [
      {
        name: "vm-example",
        computeVmWorkload: {
            instancesRunning: {
                usageRateTimeline: {
                    usageRateTimelineEntries: [
                        {
                          "usageRate": 5
                        }
                    ]
                }
            },
          machineType: {
            customMachineType: {
              machineSeries: "n1",
              virtualCpuCount: 4,
              memorySizeGb: 4
            }
          },
          region: "us-central1"
        }
      }
    ]
}

fetch(url, {
    body: requestBody
}).then(response => response.json())
```

Il risultato della richiesta contiene un riepilogo delle risorse che sono state richieste con il prezzo netto e unitario (all'ora) di ciascuna.

Per il servizio Compute Engine, va indicato il campo- **ComputeVmWorkload** in cui si specificano quante risorse fanno parte della stima del prezzo. Inoltre è necessario aggiungere la proprietà **machineType** in cui specificare una richiesta

predefinita in base al tipo standard ricavato nella fase pre-monitoraggio oppure se è una macchina personalizzata, si deve elencare le specifiche tecniche appropriate.

Altro discorso invece per il Cloud Storage in cui si deve descrivere l'oggetto **cloudStorageWorkload** indicando il numero di gigabyte occupato dagli oggetti e le operazioni effettuate sul bucket. Ovviamente le informazioni vengono raccolte tutte col processo precedente grazie alla Cloud Monitoring API.

Capitolo 5

Previsione dell'uso: implementazione

La predizione delle statistiche di utilizzo di una risorsa è implementata suddividendo l'API, esposta grazie alle funzionalità fornite da Django REST API, in due endpoint di cui uno usufruisce del modello ARIMA e l'altro che fornisce le previsioni tramite il DeepAR. Il processo tramite il quale è possibile ottenere le predizioni future sull'uso di una risorsa ha come prerequisito la collezione delle metriche. Infatti le serie temporali restituite dai vari provider con le richieste HTTP vengono inviate come payload alle interrogazioni dei due endpoint

L'implementazione della parte dell'API è la seguente:

```
class ForecastingView(APIView):
    def post(self, request):
        # Viene salvato il payload dalla richiesta
        data = request.data
        # Salvataggio dei parametri dal querystring
        query_params = request.query_params
        prediction_model = query_params.get('prediction_model')
        service_prediction = query_params.get('service_prediction')
        days_range = query_params.get('days_range')

        if prediction_model == 'arima':
            predictions = get_arima_predictions(data, prediction_service, days_range)
        elif prediction_model == 'deep_ar':
            predictions = get_deep_ar_predictions(data, prediction_service,
                days_range)

        response = predictions
        return Response(response)
```

Di seguito invece l'implementazione dell'interrogazione che viene fatta all'API tramite l'applicazione web dall'interfaccia effettuando la previsione

```
async makePrediction(datapoints: any, numberOfDays: number, service: string, model:
     string) {
    let apiEndpoint = 'http://localhost:8000/forecasting?service_prediction=${
        service}&days_range=${numberOfDays}&prediction_model=${model}'
    let response = await fetch(apiEndpoint, {
        method: "POST",
        body: datapoints
    })
    return await response.json()
}
```

Mentre l'implementazione riguardante i modelli di predizione è discussa nelle sezioni successive.

5.1 ARIMA

```
def get_arima_predictions(data, service_prediction, days_range):
    # Ordinamento dei datapoints in base al timestamp
    datapoints_sorted = sorted(data['Datapoints'], key=lambda k: k['Timestamp'])
    # Estrazione dei valori ned
    datapoints_sorted = list(map(lambda x: (x['Timestamp'], x['Value']),
        datapoints_sorted))

    # Salvataggio dei dati in un file json per poterli utilizzare in seguito
    final_data = {service_prediction: {datapoint[0]: datapoint[1] for datapoint in
        datapoints_sorted}}
    with open('datapoints_sorted.json', 'w') as f:
        json.dump(final_data, f)

    # Lettura dei dati dal file json con il parsing per creare un dataframe
    df = pd.read_json('datapoints_sorted.json')

    # Creazione del modello ARIMA a partire dal dataframe
    auto_model = pm.auto_arima(df, stepwise=False, seasonal=False)

    if days_range:
        days_range = int(days_range)

    # Ottiene la data finale del range a partire dal numero di giorni
    new_date = (datetime.datetime.now() + datetime.timedelta(days=days_range)).
        isoformat()

    # Crea un indice di date a partire dalla data attuale fino alla data finale
    index_future_dates = pd.date_range(start=datetime.datetime.now().isoformat(),
        end=new_date, freq='H')

    # Esegue la predizione del modello ARIMA sulle date future
    auto_model_predictions = auto_model.predict(n_periods=len(index_future_dates)).
        rename('Auto ARIMA Predictions')
    auto_model_predictions.index = index_future_dates

    # Plot dei risultati
    auto_model_predictions.plot(legend=True)
    df[service_prediction].plot(legend=True)

    plt.show()

    return auto_model_predictions
```

5.2 DeepAR

```
import json

import lightning.pytorch as pl
from lightning.pytorch.callbacks import EarlyStopping
import matplotlib.pyplot as plt
import pandas as pd
import warnings
from pytorch_forecasting import Baseline, DeepAR, TimeSeriesDataSet
from pytorch_forecasting.data import NaNLabelEncoder
```

```
from pytorch_forecasting.metrics import MAE, SMAPE,
    MultivariateNormalDistributionLoss
from lightning.pytorch.tuner import Tuner
import torch

def get_deep_ar_predictions(data, service_prediction, days_range):
    # Ordinamento dei datapoints in base al timestamp
    datapoints_sorted = sorted(data['Datapoints'], key=lambda k: k['Timestamp'])
    datapoints_sorted = list(map(lambda x: (x['Timestamp'], x['Average']),
        datapoints_sorted))

    final_data = {service_prediction: {idx: datapoint[1] for idx, datapoint in
        enumerate(datapoints_sorted)}, 'date': {idx: datapoint[0] for idx,
        datapoint in enumerate(datapoints_sorted)},
                "group": {idx: 0 for idx, _ in enumerate(datapoints_sorted)}}
    with open('datapoints_sorted.json', 'w') as f:
        json.dump(final_data, f)

    train_df = pd.read_json('datapoints_sorted.json')

    # Aggiunge valori NaN alla colonna service_prediction per le date specificate
    min_date = train_df["date"].min()
    train_df["time_idx"] = train_df["date"].map(lambda current_date: int((
        current_date - min_date).total_seconds() / 3600))


    max_encoder_length = 350
    max_prediction_length = 150

    # Il training cutoff e' il giorno a partire dal quale non vengono piu'
        utilizzati i datapoints per il training
    training_cutoff = train_df["time_idx"].max() - max_prediction_length

    # Il dataset da utilizzare per il training e' composto da tutti i datapoints
        fino al training cutoff
    training = TimeSeriesDataSet(
        train_df[lambda x: x.time_idx <= training_cutoff],
        time_idx="time_idx",
        target=service_prediction,
        group_ids=["group"],
        categorical_encoders={service_prediction: NaNLabelEncoder().fit(train_df.
            CPUUtilization)},
        time_varying_unknown_reals=[service_prediction],
        max_encoder_length=max_encoder_length,
        max_prediction_length=max_prediction_length,
        allow_missing_timesteps=True
    )

    # Il dataset da utilizzare per la validazione  composto da tutti i datapoints a
        partire dal training cutoff
    validation = TimeSeriesDataSet.from_dataset(training, train_df,
        min_prediction_idx=training_cutoff + 1)
    batch_size = 128

    # Creazione dei dataloaders per il training e la validazione
    train_dataloader = training.to_dataloader(
        train=True, batch_size=batch_size, num_workers=0
    )
    val_dataloader = validation.to_dataloader(
        train=False, batch_size=batch_size, num_workers=0
    )

    pl.seed_everything(42)
```

```
# Caricamento del modello DeepAR
loaded_model = torch.load(f'{service_prediction}.pth')

# Se il modello non  stato salvato precedentemente, viene creato e salvato
if not loaded_model:
    import pytorch_forecasting as ptf

    trainer = pl.Trainer(accelerator="cpu", gradient_clip_val=1e-1)
    net = DeepAR.from_dataset(
        training,
        learning_rate=3e-2,
        hidden_size=30,
        rnn_layers=2,
        loss=MultivariateNormalDistributionLoss(rank=30),
        optimizer="Adam",
    )

    res = Tuner(trainer).lr_find(
        net,
        train_dataloaders=train_dataloader,
        val_dataloaders=val_dataloader,
        min_lr=1e-5,
        max_lr=1e0,
        early_stop_threshold=None,
    )


    early_stop_callback = EarlyStopping(monitor="val_loss", min_delta=1e-4,
         patience=10, verbose=False, mode="min")
    trainer = pl.Trainer(
        max_epochs=30,
        accelerator="cpu",
        enable_model_summary=True,
        gradient_clip_val=0.1,
        callbacks=[early_stop_callback],
        limit_train_batches=50,
        enable_checkpointing=True,
    )

    net = DeepAR.from_dataset(
        training,
        learning_rate=5e-3,
        log_interval=10,
        log_val_interval=1,
        hidden_size=30,
        rnn_layers=2,
        optimizer="Adam",
        loss=MultivariateNormalDistributionLoss(rank=30),
    )

    trainer.fit(
        net,
        train_dataloaders=train_dataloader,
        val_dataloaders=train_dataloader,
    )


    best_model_path = trainer.checkpoint_callback.best_model_path
    best_model = DeepAR.load_from_checkpoint(best_model_path)

    torch.save(best_model, f'{service_prediction}.pth')

# Esecuzione della predizione
raw_predictions, x, index, decoder_lengths, y = loaded_model.predict(
```

```
        val_dataloader, mode="raw", return_x=True,
        return_decoder_lengths=True,
        return_index=True,
        return_y=True,
        trainer_kwargs=dict(accelerator="cpu"),
    )

    loaded_model.plot_prediction(x, raw_predictions, idx=0, add_loss_to_title=True)

    plt.show()

    return raw_predictions
```

Capitolo 6

Test e conclusioni

6.1 Test

In questa sezione viene mostrata l'interfaccia interattiva della web app che interagisce con le API dei vari servizi cloud e con quella in loco che si occupa di prevedere l'utilizzo.

Figura 6.1. Landing page con scelta del provider

Per ogni servizio di seguito viene illustrato come all'utente finale appaiono le richieste delle varie funzionalità offerte dal Price Broker.

6.1.1 Visualizzazione Istanze, raccolta metriche e calcolo dei costi Amazon Web Services

Figura 6.2. Lista dei servizi

Figura 6.3. Lista delle istanze

Figura 6.4. Scelta dell'istanza

Figura 6.5. Calcolo dei costi

Microsoft Azure

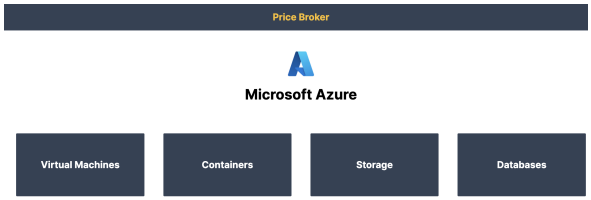

Figura 6.6. Lista dei servizi

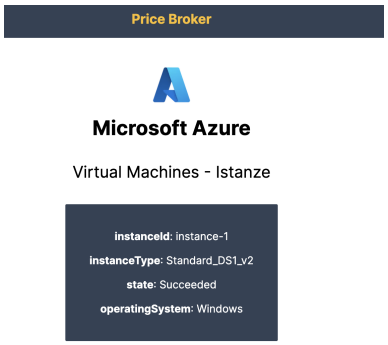

Figura 6.7. Lista dell'istanze

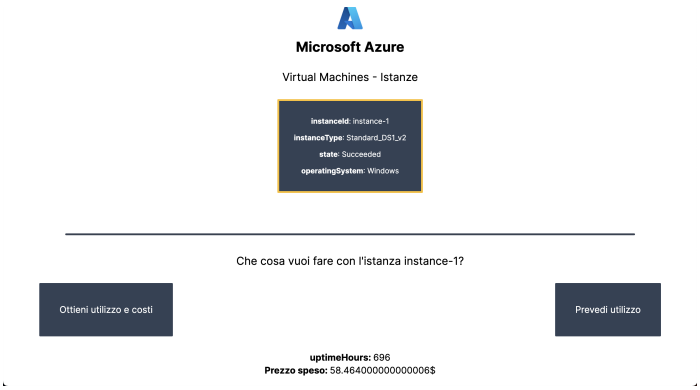

Figura 6.8. Calcolo dei costi

Google Cloud

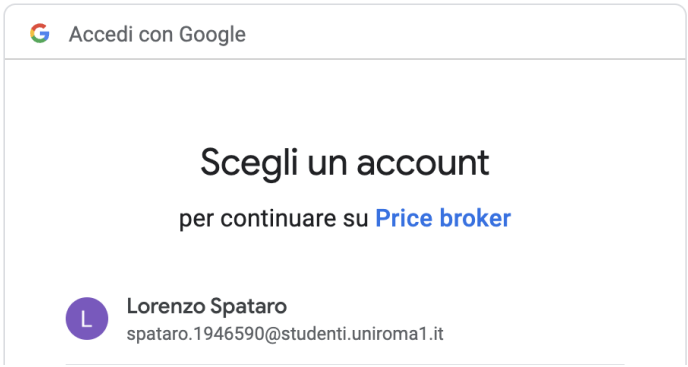

Figura 6.9. Autenticazione

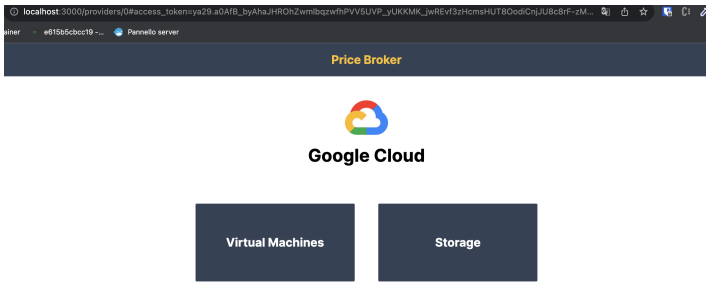

Figura 6.10. Scelta del servizio

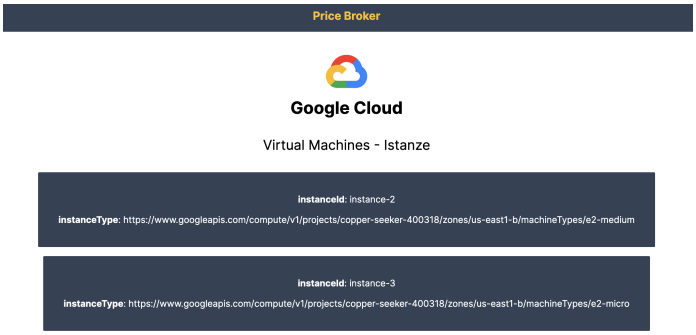

Figura 6.11. Lista delle istanze

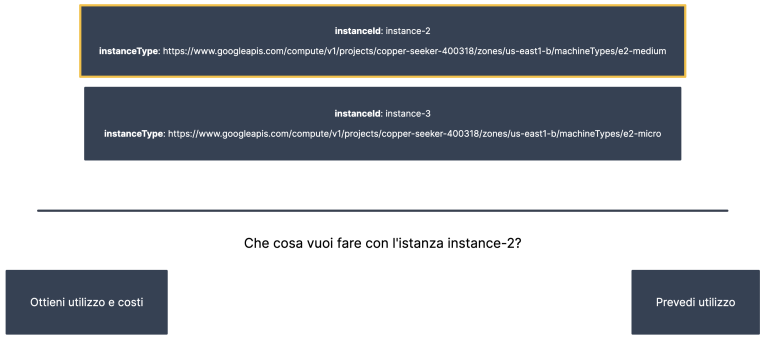

Figura 6.12. Scelta dell'istanza

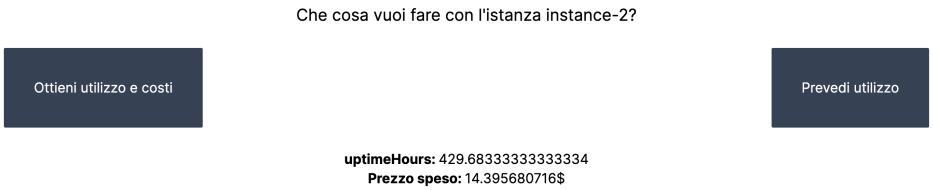

Figura 6.13. Avvio e risultato del calcolo dei costi

6.1.2 Predizioni dell'uso

Per questo test è stato preso in considerazione l'utilizzo della CPU di una delle istanze EC2 di AWS. I datapoint che vengono ricevuti da CloudWatch vengono inviati sottoforma di payload all'API che li elabora, costruisce e allena i modelli se necessario ed inseguito restituisce i datapoint della predizione.
Di seguito i grafici per ogni modello che mostrano le predizioni nei 6 giorni successivi alla collezione delle statistiche.

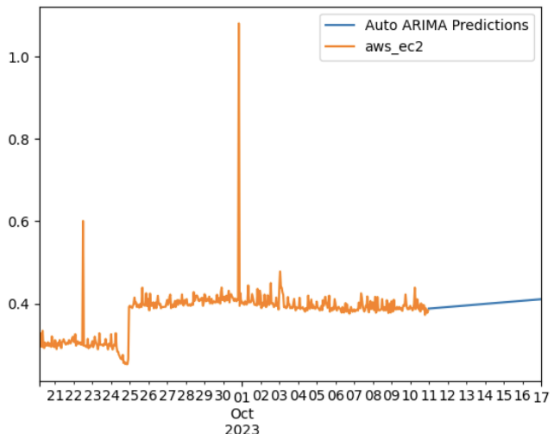

Figura 6.14. Predizione con modello ARIMA

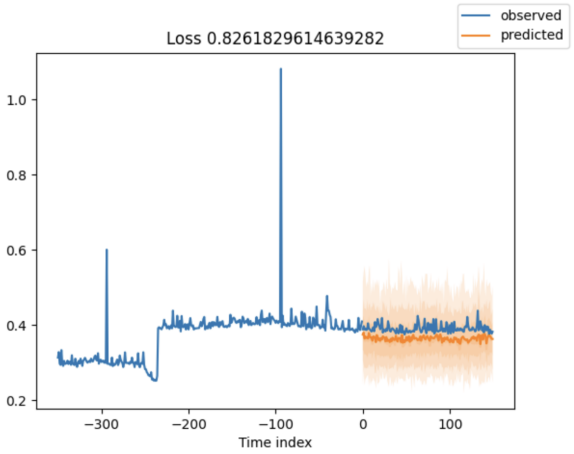

Figura 6.15. Predizione con modello DeepAR

6.1.3 Conclusioni

Il tirocinio si è concentrato sulla progettazione e implementazione di un innovativo servizio di price broker per il calcolo ed il confronto dei costi di servizi cloud. L'obiettivo principale era sviluppare una soluzione in grado di raccogliere dati accurati, stimare i costi corrispondenti e fornire previsioni affidabili sull'uso futuro dei servizi cloud.

Riepilogo degli Obiettivi

Gli obiettivi iniziali di questa ricerca erano chiari: implementare un price broker efficace, in grado di affrontare le sfide della gestione dei costi nei servizi cloud. Attraverso l'analisi dettagliata delle metriche di utilizzo e l'applicazione di algoritmi predittivi avanzati, abbiamo cercato di fornire agli utenti uno strumento completo per ottimizzare le spese cloud.

Risultati e Contributi

I risultati ottenuti durante le fasi di testing e validazione sono promettenti. I contributi principali di questa tesi includono la progettazione di un'architettura flessibile, l'implementazione di algoritmi predittivi robusti e la creazione di un'interfaccia utente intuitiva per la visualizzazione delle risorse e analisi delle metriche.

Applicazioni Potenziali

Le applicazioni pratiche di questo servizio sono ampie e diversificate. Le aziende possono utilizzare il price broker per ottimizzare le loro risorse, prendere decisioni informate sulla scalabilità dei servizi e ridurre i costi operativi associati all'uso dei servizi cloud.

Sviluppi Futuri

Guardando al futuro, ci sono diverse direzioni che potrebbero essere esplorate per migliorare ulteriormente il servizio. Possibili sviluppi futuri includono l'integrazione di nuove metriche di utilizzo, l'ottimizzazione degli algoritmi predittivi e l'estensione del supporto per più provider di servizi cloud e per una gamma maggiore di servizi.

6.1.4 Conclusioni Finali

In conclusione, questa relazione ha fornito una solida base per affrontare le sfide legate alla gestione dei costi nei servizi cloud. Il price broker rappresenta un passo significativo verso una gestione più efficiente delle risorse cloud, offrendo agli utenti uno strumento pratico e affidabile per ottimizzare i loro investimenti.

Bibliografia

[1] Casalicchio, E., Cardellini, V., Interino, G., and Palmirani, M. Research challenges in legal-rule and qos-aware cloud service brokerage. *Future Generation Computer Systems*, **78** (2018), 211. Available from: https://www.sciencedirect.com/science/article/pii/S0167739X16306641, doi:https://doi.org/10.1016/j.future.2016.11.025.

[2] Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., and Leaf, D. NIST cloud computing reference architecture. Tech. rep., U.S. Department of Commerce (2011). Available from: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-292.pdf.

[3] Mell, P. and Grance, T. The NIST definition of cloud computing. Tech. rep., U.S. Department of Commerce (2011). Available from: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[4] Vakili, M., Jahangiri, N., and Sharifi, M. Cloud service selection using cloud service brokers: approaches and challenges. *Frontiers of Computer Science*, (2018). Available from: https://link.springer.com/article/10.1007/s11704-017-6124-7, doi:https://doi.org/10.1007/s11704-017-6124-7.