

Corso di Laurea in Ingegneria Informatica M

FantaNBA:

Servizio per la predizione di partite NBA

Relazione Progetto Scalable And Reliable Services

Repository: <https://github.com/thomasBianconcini/ScalableProjectNBA>

Presentata da:
Alberto Garagnani
Leonardo Di Fabio
Thomas Bianconcini
Tommaso Muzzi

Indice

Introduzione	2
1 Tecnologie Utilizzate	3
1.1 Microsoft Azure	3
1.2 Flask	3
1.3 Node & React	4
1.4 Docker	4
1.5 Kubernetes	4
1.6 Mongo DB	5
1.7 TensorFlow & Keras	5
1.8 Locust	5
2 Struttura dell'applicativo	6
2.1 Frontend	7
2.2 Backend	8
2.2.1 Manipolazione dei Dati Input	8
2.2.2 Rete Neurale	9
2.2.3 API	10
2.2.4 Database	11
2.2.5 Predict	12
3 Infrastruttura Generale	13
3.1 Docker	14
3.2 ACR - Azure Container Registry	14
3.3 AKS - Azure Kubernetes Service	14
3.4 Azure Cosmos DB for Mongo DB	15
4 Scalabilità, Affidabilità e Sicurezza	16
4.1 Scalabilità automatica in Azure	16
4.2 Repliche dei Pod	17
4.3 HTTPS	17
4.4 Test	19

Introduzione

Il presente report esplora lo sviluppo di un applicativo web per la predizione dei risultati delle gare sportive. L'applicazione è concepita per appassionati e analisti del basket, e permette la selezione di 2 squadre del campionato NBA, restituendo una predizione del risultato mediante un sistema di intelligenza artificiale. L'applicativo gira sul servizio cloud Azure, e, mediante esso, soddisfa i requisiti di scalabilità, affidabilità e sicurezza imposti dall'informatica moderna. Questo strumento si propone di trasformare il modo in cui appassionati e tifosi interagiscono con il basket.

Capitolo 1

Tecnologie Utilizzate

Nel capitolo successivo, esploreremo una serie di tecnologie fondamentali che costituiscono l'infrastruttura e gli strumenti chiave per lo sviluppo e l'implementazione dell'applicativo. Queste tecnologie rappresentano pilastri essenziali per la creazione di un sistema robusto e scalabile che integra l'analisi dei dati e le previsioni avanzate con un'interfaccia utente moderna e reattiva. In particolare, analizzeremo come Azure, una piattaforma leader nel cloud computing, fornisca l'infrastruttura necessaria per ospitare e gestire il nostro ambiente di sviluppo e produzione. Esploreremo inoltre Flask e Node.js, due framework leggeri e potenti per lo sviluppo di API e applicazioni web, essenziali per il nostro servizio backend e per l'interfaccia utente dinamica. Continueremo con Docker e Kubernetes, che rivestono un ruolo cruciale nella gestione dei container e nell'orchestrazione delle nostre applicazioni containerizzate. Queste tecnologie ci permettono di garantire una distribuzione affidabile e scalabile del nostro servizio su ambienti cloud multipli.

MongoDB sarà discusso come sistema di gestione dei database flessibile e scalabile, utilizzato per archiviare e gestire i dati necessari per le previsioni e le analisi dell'intelligenza artificiale. Infine, esploreremo TensorFlow, un framework di machine learning open-source, essenziale per l'implementazione dei modelli predittivi avanzati all'interno del nostro servizio.

1.1 Microsoft Azure

Microsoft Azure è una piattaforma cloud completa che offre una vasta gamma di servizi di cloud computing. È grazie ad Azure che è stato possibile ospitare il servizio online, fornendo le risorse necessarie per la sua realizzazione e gestione.

1.2 Flask

Flask è un framework web leggero e estensibile per Python, progettato per sviluppare applicazioni web in modo semplice e veloce. La possibilità di offrire API con Flask è una delle caratteristiche principali e più utilizzate di questo framework. Flask facilita la creazione di

API RESTful grazie a Flask-RESTX, consentendo agli sviluppatori di esporre funzionalità del Backend attraverso endpoint HTTP standard come GET, POST, ecc. Flask è stato utilizzato per la gestione del servizio Backend e per l'esposizione dell'API relativa alla predizione delle partite NBA.

1.3 Node & React

Node.js è un runtime JavaScript server-side che consente di eseguire codice JavaScript al di fuori del browser. È ampiamente utilizzato per gestire dipendenze, creare server di sviluppo, implementare server-side rendering e sviluppare microservizi e API per il backend delle applicazioni.

React invece, è una libreria JavaScript per la creazione di interfacce utente, sviluppata da Facebook. È stato utilizzato React nel frontend del nostro servizio per costruire componenti riutilizzabili e gestire lo stato dell'interfaccia utente in modo efficiente, migliorando la reattività e le prestazioni complessive dell'applicazione web con il supporto di Node.js come runtime System.

1.4 Docker

Docker è una piattaforma di containerizzazione che consente di creare, distribuire e gestire applicazioni in ambienti isolati chiamati container. I container Docker includono il codice dell'applicazione e tutte le sue dipendenze, garantendo che l'applicazione funzioni in modo uniforme e prevedibile su qualsiasi ambiente, indipendentemente dalle differenze tra sistemi operativi e configurazioni di hosting.

Sia il frontend React che il backend Flask sono stati inglobati in due container docker distinti sia per garantire l'isolamento degli ambienti sia per maggiore portabilità.

1.5 Kubernetes

Kubernetes è un sistema open-source per l'automazione del deployment, della scalabilità e della gestione di applicazioni containerizzate. È progettato per gestire in modo efficiente e automatizzato le operazioni su larga scala di container Docker, garantendo la disponibilità, la scalabilità e la gestione delle risorse delle applicazioni.

I container sopra citati sono stati inglobati all'interno di due pod Kubernetes.

1.6 Mongo DB

MongoDB è un database NoSQL orientato ai documenti, progettato per facilitare lo sviluppo e la scalabilità delle applicazioni. A differenza dei tradizionali database relazionali, MongoDB utilizza strutture simili a JSON chiamate documenti, che possono avere schemi variabili. Questo lo rende particolarmente flessibile e adatto a gestire grandi volumi di dati strutturati, semi-strutturati e non strutturati con rapidità e facilità.

MongoDB è stato utilizzato per immagazzinare le informazioni dei singoli giocatori di NBA.

1.7 TensorFlow & Keras

TensorFlow è una delle principali librerie in ambito Intelligenza Artificiale e offre un grande numero di possibili implementazioni di reti neurali e altri strumenti per il Machine Learning e Deep Learning. Keras è una libreria che è ormai altamente collegata a TensorFlow e fornisce metodi per la creazione e manipolazione delle reti che semplificano di molto il processo di sviluppo. TensorFlow è stato utilizzato per tutta la componente di Machine Learning del progetto.

1.8 Locust

Locust è uno strumento open-source per il testing delle prestazioni e il carico di lavoro delle applicazioni web. è progettato per simulare il traffico utente e misurare come le applicazioni rispondono sotto diversi livelli di carico. Locust è stato utilizzato per la fase di testing.

Capitolo 2

Struttura dell'applicativo

Nel seguente capitolo viene esaminata la struttura dell'applicazione, composta da un frontend React e un backend API. Il frontend React permette agli utenti di inserire le informazioni relative a due squadre e richiedere una predizione sul vincitore, offrendo un'interfaccia intuitiva e interattiva. Il backend espone una singola API, la quale sfrutta le capacità dell'intelligenza artificiale per predire il risultato dell'incontro tra le squadre specificate. Entrambi i servizi risiedono all'interno di container docker, garantendo così un deployment più semplice e un ambiente di esecuzione più controllato. Questa architettura combina la potenza del machine learning con l'accessibilità e l'usabilità di un'interfaccia utente, offrendo agli utenti una previsione informata basata su dati analitici avanzati.

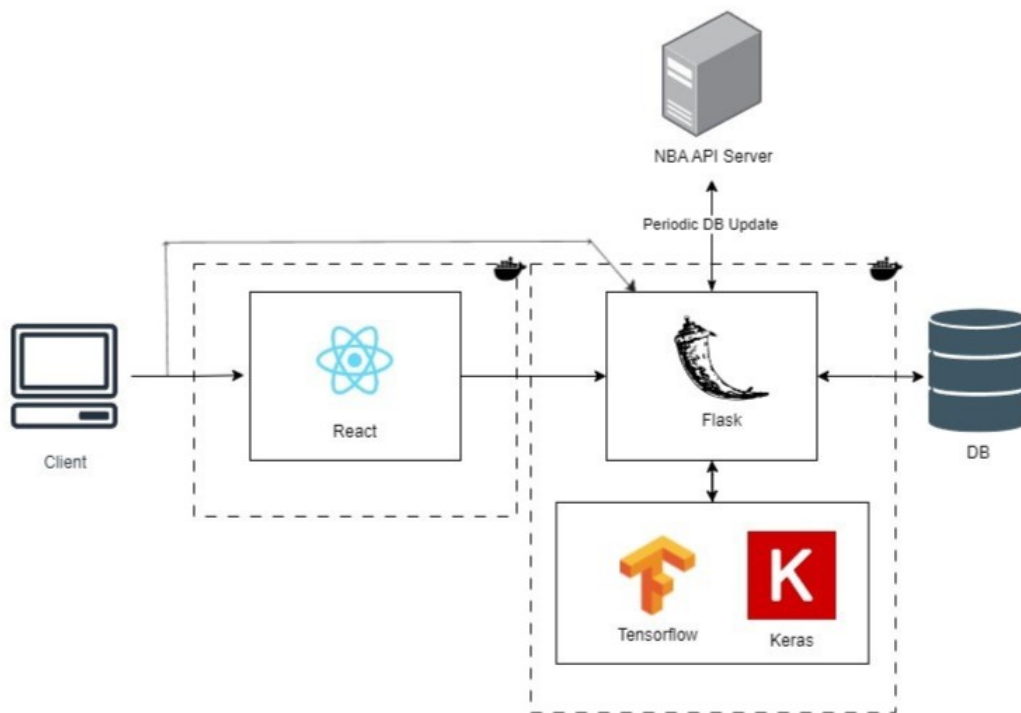


Figura 4: Schema Struttura Applicativo

2.1 Frontend

Il servizio frontend è stato sviluppato in React, presenta una sola pagina con al suo interno la possibilità di scegliere due squadre e ottenere una predizione del vincente tramite un pulsante. Utilizza diverse librerie e componenti per la gestione dell'interfaccia utente e delle chiamate API.

- `useState`: Hook di React per gestire lo stato locale all'interno del componente.
- `Material-UI`: Fornisce componenti UI come `Button`, `Grid`, e `Typography` per il layout e l'aspetto dell'applicazione.
- `react-loader-spinner`: Componente per mostrare un indicatore di caricamento animato.
- `BasicSelect`: Componente personalizzato per selezionare le squadre.
- `ButtonAppBar`: Componente personalizzato per la barra di navigazione.

Nello stato sono gestite le variabili `squadra1`, `squadra2` e `value` che sono rispettivamente le due squadre selezionate e il valore ottenuto.

```
const [squadra1, setSquadra1] = useState("");
const [squadra2, setSquadra2] = useState("");
const [value, setValue] = useState("");
```

Ad ogni componente `BasicSelect` per la selezione di una squadra è associato un metodo `changeTeam` che permette di salvare nello stato il valore selezionato.

```
const changeTeam1 = (team: string) => {
  setSquadra1(team);
}
```

In questo modo, quando un utente seleziona una squadra, React salva il valore in memoria nello stato per poterlo utilizzare successivamente nella richiesta API.

Questa richiesta viene gestita tramite una `fetch` di React, i cui parametri necessari sono:

- `backendIp`: Indirizzo IP dell'API presente nel servizio backend.
- `apiUrl`: Indirizzo IP sulla quale effettuare la `fetch`, al suo interno presenta il `backendIP` con la rispettiva route dell'API e i nomi delle due squadre presenti nello state.

```
https://${backendIp}/api/predict?squadra1=${squadra1}
&squadra2=${squadra2}
```

- `apiKey`: Chiave di accesso per la sicurezza dell'API.
- `Access-Control-Allow-Origin`: Necessario per la gestione di TLS.

Quando questa fetch restituisce un valore, il risultato viene salvato nello stato e mostrato a schermo come output della richiesta del vincitore della partita.

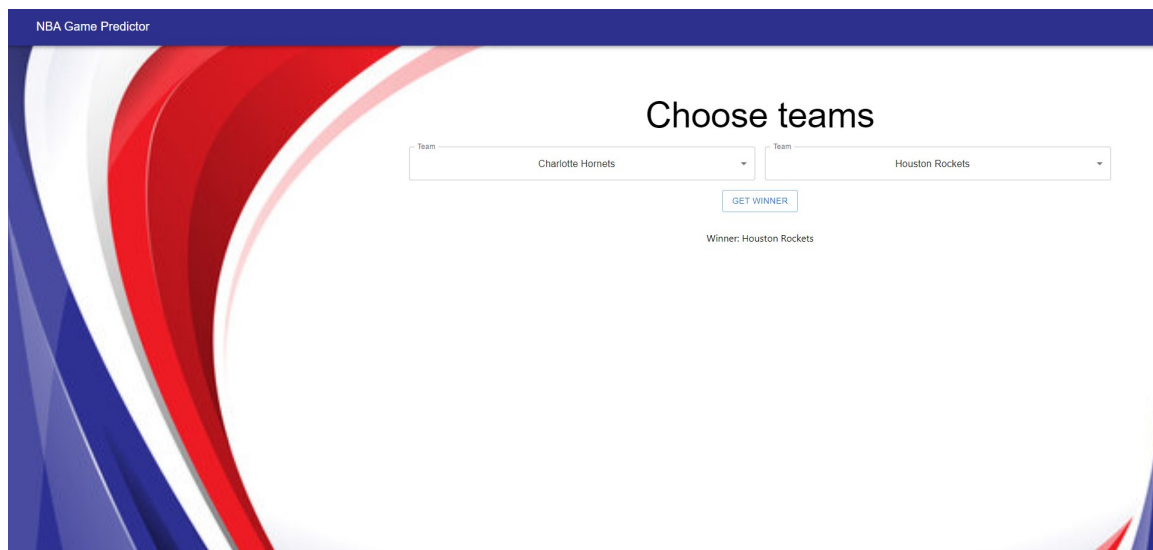


Figura 1: Immagine frontend.

2.2 Backend

Il servizio backend utilizza Flask e Flask-Restx per creare un'API che fornisce previsioni di risultati per partite di basket NBA.

2.2.1 Manipolazione dei Dati Input

Per la gestione dei dati necessari all'allenamento della rete delle partite NBA, è stato impiegato il servizio nba_api, una libreria che permette di interrogare il sito ufficiale della NBA per acquisire varie tipologie di informazioni.

Durante la fase di training, sono stati raccolti i dati storici delle partite degli ultimi tre anni, andando a ottenere dettagli sulle squadre e sugli esiti delle partite. Successivamente, per ogni giocatore delle due squadre, sono state estratte le statistiche relative al periodo della partita considerata. Questi dati sono stati poi aggregati sommando le statistiche individuali dei giocatori di ciascuna squadra, ottenendo un array di 15 valori per squadra, per un totale di 30 valori. Prima di utilizzare questi dati come input per la rete neurale, è stata applicata una standardizzazione mediante l'uso di uno StandardScaler, garantendo così la normalizzazione dei dati.

Per la fase di predizione, è stato seguito lo stesso processo di acquisizione e trattamento dei dati, con la standardizzazione che ha impiegato il modello di StandardScaler salvato precedentemente. Anche il modello predittivo è stato precedentemente salvato e caricato per effettuare le predizioni.

```
for i, row in history_df.iterrows():
```

```

home_team = row[ 'Home_Team' ]
away_team = row[ 'Away_Team' ]
teams_query= [home_team, away_team]
season = row[ 'Season' ]

array_1 = np.array([0.0] * num_stats)
array_2 = np.array([0.0] * num_stats)
teams_stats = [array_1, array_2]

for idx, team in enumerate(teams_query):
    t1 = teams.find_teams_by_full_name(team)
    team_id = t1[0][ 'id' ]

    t = teamplayerdashboard.TeamPlayerDashboard(team_id=team_id,
                                                  season=season)
    data_player = json.loads(t.get_json())[ 'resultSets' ]
    players_data = next(item for item in data_player if
                        item[ 'name' ] == 'PlayersSeasonTotals')[ 'rowSet' ]
    player_ids = [player[1] for player in players_data]

    for player_id in player_ids:
        career = playercareerstats.PlayerCareerStats
                    (player_id=player_id)
        df_single_player_carrer = career.get_data_frames()
        [0] df_single_player_carrer.columns.get_loc
                    ('PLAYER_AGE') + 1:].tolist()
        last_row_stats = df_single_player_carrer[selected_columns]
                    .iloc[-1].tolist()
        teams_stats[idx] += np.array(last_row_stats)

```

2.2.2 Rete Neurale

La rete neurale è stata sviluppata utilizzando TensorFlow e Keras. Per la predizione sull'esito della partita è stata implementata una rete MLP (Multilayer Perceptron) che è fully connected, questo significa che ogni neurone di uno strato della rete è connesso con tutti i nodi successivi dello strato seguente. La rete è un classificatore binario, deve infatti soltanto decidere se il vincitore è la squadra in casa o la squadra in trasferta.

La rete è stata strutturata nel seguente modo:

- **Strato Input:** l'input è formato da 30 nodi che rappresentano le statistiche sommate di ogni giocatore (15 per la squadra in casa, 15 per la squadra fuori casa).
- **Strati Nascosti:** sono presenti 2 strati nascosti, ognuno dei quali è formato da 512 nodi. Come funzione di attivazione dei 2 strati è stata utilizzata la reLU che è considerata tra le migliori funzioni nei casi di reti fully connected.
- **Strato Output:** siccome abbiamo un solo valore da trovare lo strato di output è formato da un solo nodo. Dal momento che il classificatore è binario è stato scelto di utilizzare come funzione di attivazione la sigmoide, che meglio si presta a queste casistiche.
- **Compile:** come valori con cui compilare la rete si è scelto adam come optimizer visto che era quello che lavorava meglio nel nostro caso, come loss function invece è stata utilizzata la "binary_crossentropy" che è la funzione migliore nei casi di classificatori binari.

```
input_shape = (Xtrain_std.shape[1],)
```

```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=input_shape))
model.add(Dense(512, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

```
adam = Adam(learning_rate=0.001)
```

```
model.compile(optimizer=adam, loss='binary_crossentropy',
              metrics=['accuracy'])
```

2.2.3 API

Le librerie utilizzate sono:

- Flask-RESTX: Una estensione di Flask che semplifica la creazione di API RESTful.
- json, request, jsonify: Moduli di Flask per gestire dati JSON e richieste HTTP.
- Namespace: Una classe di Flask-Restx per definire uno spazio dei nomi per le API.
- leaguegamefinder: Un endpoint della libreria nba_api.stats.endpoints per trovare partite nella lega NBA.
- datetime: Per operazioni relative alle date e orari.

Tramite `@api.route("/predict")` viene dichiarato un endpoint `/predict` per l'API. Al suo interno viene specificato l'apikey necessario per la sicurezza. L'endpoint riceve due parametri (`Squadra1`, `Squadra2`) che vengono dati come parametri al metodo `predict` che si occuperà di ritornare il valore della predizione.

```
@api.route("/predict")
class HelloWorld(Resource):
    @api.doc(security='apikey')
    @api.expect(api_key)
    def get(self):
        api_key = request.headers.get('X-API-KEY')
        if not check_api_key(api_key):
            return {'message': 'Chiave API non valida'}, 401
        squadra1 = request.args.get('squadra1')
        squadra2 = request.args.get('squadra2')
        return predict(squadra1, squadra2)
```

2.2.4 Database

E' stato generato un database per evitare di interrogare la api di nba troppe volte inutilmente. Il database è hostato all'interno di un Azure Cosmos DB for MongoDB (vCore) e presenta al suo interno tutte le statistiche della stagione corrente dei giocatori NBA.

Il database si presenta come segue:

	_id	TEAM_ID	PLAYER_ID	GP	GS	MIN	FG_PCT	FG3_PCT	FT_PCT	OREB	DREB	REB	AST	STL	BLK	TOV	PF	PT
0	666ac8f7a6184f1aa3543168	1610612737	1631100	20.0	0.0	171.0	0.290	0.256	1.000	2.0	16.0	18.0	5.0	1.0	2.0	8.0	6.0	48.
1	666ac8f7a6184f1aa3543169	1610612737	203992	79.0	33.0	2401.0	0.428	0.374	0.921	53.0	219.0	272.0	247.0	96.0	26.0	112.0	179.0	1333.
2	666ac8f7a6184f1aa354316a	1610612737	1628981	45.0	2.0	682.0	0.583	0.000	0.667	59.0	134.0	193.0	46.0	25.0	28.0	44.0	107.0	282.
3	666ac8f7a6184f1aa354316b	1610612737	203991	73.0	73.0	1883.0	0.571	0.000	0.631	335.0	441.0	776.0	91.0	43.0	106.0	74.0	164.0	836.
4	666ac8f7a6184f1aa354316c	1610612737	1629631	57.0	37.0	1681.0	0.459	0.385	0.847	31.0	191.0	222.0	87.0	40.0	16.0	83.0	146.0	891.
...
652	666ac8f7a6184f1aa35433f4	1610612764	1626158	40.0	10.0	554.0	0.558	0.333	0.723	71.0	111.0	182.0	25.0	11.0	19.0	24.0	74.0	199.
653	666ac8f7a6184f1aa35433f5	1610612764	1631157	13.0	0.0	78.0	0.519	0.750	0.765	2.0	11.0	13.0	14.0	10.0	3.0	8.0	8.0	44.
654	666ac8f7a6184f1aa35433f6	1610612764	1641998	25.0	14.0	574.0	0.551	0.000	0.840	64.0	70.0	134.0	27.0	12.0	28.0	34.0	69.0	171.
655	666ac8f7a6184f1aa35433f7	1610612764	1641774	10.0	4.0	153.0	0.433	0.278	0.773	6.0	30.0	36.0	13.0	5.0	7.0	10.0	28.0	85.
656	666ac8f7a6184f1aa35433f8	1610612764	1626145	66.0	66.0	1933.0	0.489	0.414	0.800	20.0	159.0	179.0	485.0	71.0	18.0	66.0	48.0	790.

657 rows x 18 columns

Figura 2: Database statistiche giocatori.

Flask per accedere il db utilizza la connection string fornita da Azure per accedere ai dati contenuti al suo interno. Tramite un metodo `_get_data_from_db` è possibile ricevere i dati relativi ad una squadra necessari per il modello.

2.2.5 Predict

Il metodo `predict` si occupa di rispondere alla richiesta API con l'output della predizione, riceve in input i nomi delle due squadre ricevuti nell'endpoint Flask `/predict`. Accede di conseguenza al database tramite il metodo `_get_data_from_db` per ricevere le informazioni delle due squadre.

```
home_team_players = _get_data_from_db (TEAM_IDS[home])  
away_team_players = _get_data_from_db (TEAM_IDS[away])
```

Viene infine creato un dataframe formattato correttamente, con al suo interno le informazioni della squadra di casa e della squadra ospite necessarie per il modello. Tramite l'utilizzo di tensorflow viene invocato il modello per ottenere in output la squadra vincente e redirezionarla in uscita all'API.

Infrastruttura Generale

Il progetto è stato realizzato su Microsoft Azure, utilizzando diversi servizi messi a disposizione dalla piattaforma cloud. Il servizio sviluppato è basato su container, in cui il frontend e il backend sono stati integrati in due immagini Docker separate. Queste immagini sono state appositamente curate e caricate in un Azure Container Registry (ACR), un repository dedicato alla gestione sicura delle immagini container. L'utilizzo delle immagini da ACR facilita l'implementazione su Kubernetes, consentendo una distribuzione efficiente delle applicazioni. Questo capitolo esplora l'architettura del sistema, mettendo in luce le scelte progettuali e le strategie di implementazione adottate per garantire prestazioni ottimali e scalabilità nel contesto dell'ambiente cloud.

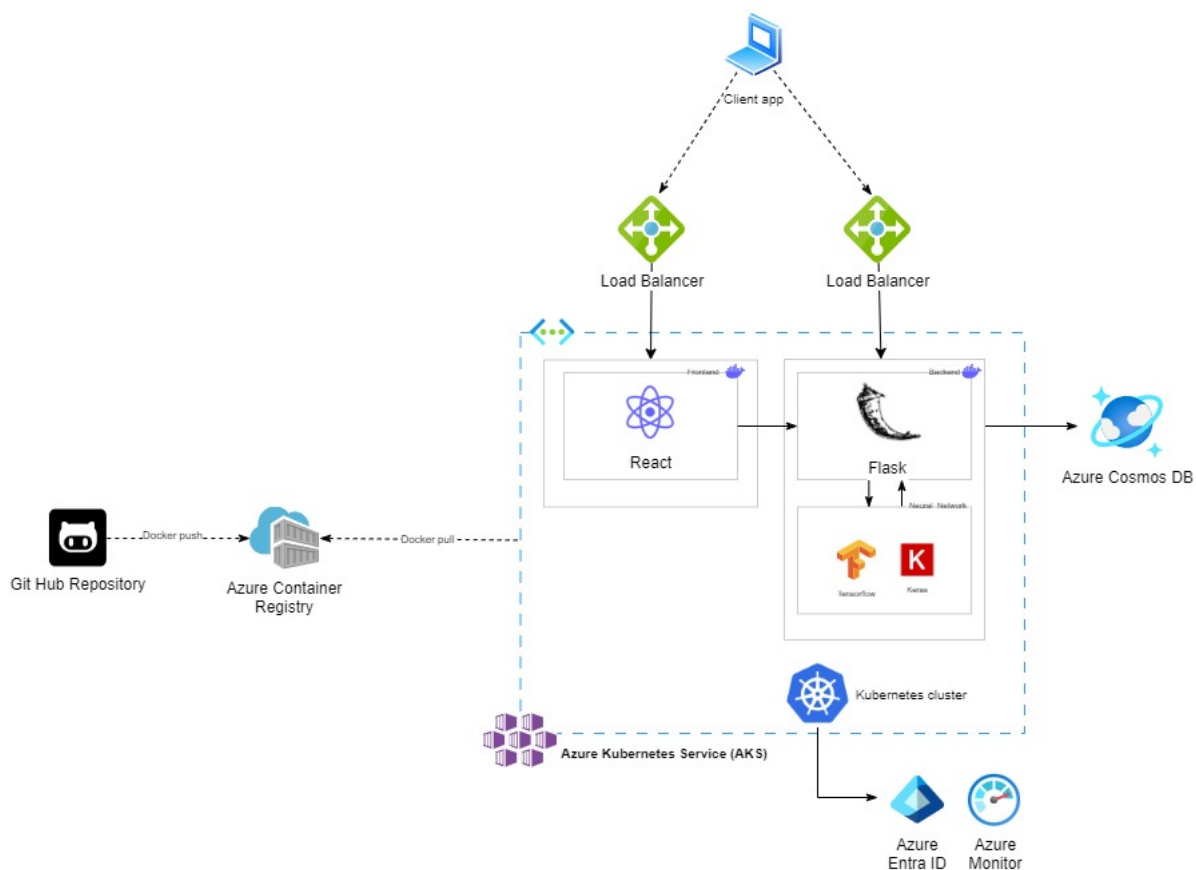


Figura 3: Infrastruttura.

3.1 Docker

Tramite l'utilizzo di Docker sono state create due immagini: una per il servizio front-end e una per il servizio backend dell'applicazione. Queste immagini contengono tutti i componenti necessari per l'esecuzione dei rispettivi servizi, inclusi codice dell'applicazione, dipendenze, configurazioni e librerie. Tramite l'utilizzo di un file `docker-compose.yaml` è stata definita la configurazione per l'esecuzione degli ambienti di sviluppo e di produzione, includendo dettagli come la connessione TLS e il binding delle porte. Questo file YAML permette di orchestrare l'avvio e la configurazione di più container Docker come un'applicazione distribuita, garantendo che ogni container abbia accesso alle risorse necessarie e sia configurato in modo appropriato per la comunicazione sicura e efficace tra i servizi.

3.2 ACR - Azure Container Registry

Azure Container Registry (ACR) consente di archiviare in modo sicuro e gestire immagini dei container Docker. È progettato per supportare applicazioni containerizzate su piattaforma Azure, offrendo integrazione nativa con Kubernetes e altri servizi cloud. Azure Container Registry (ACR) è utilizzato per archiviare e gestire le immagini dei container Docker, inclusi i container che compongono il front end e il back end di un'applicazione. Questo processo di integrazione dei container Docker con ACR è fondamentale per gestire in modo efficiente e sicuro il ciclo di vita delle applicazioni containerizzate su piattaforma Azure. Dopo il caricamento delle immagini dei container su ACR, queste vengono utilizzate all'interno di Kubernetes per eseguire i deployment delle applicazioni.

3.3 AKS - Azure Kubernetes Service

AKS si occupa dell'implementazione, della gestione e della manutenzione dei cluster Kubernetes. AKS offre scalabilità orizzontale e verticale dei nodi del cluster Kubernetes per gestire carichi di lavoro dinamici e crescenti. Inoltre, gestisce la disponibilità e la ridondanza dei nodi per garantire l'affidabilità delle applicazioni.

È stato configurato un cluster Kubernetes che include due load balancer interni per gestire il traffico del servizio. Ogni load balancer è stato configurato con due pod, progettati per garantire un'elevata affidabilità e disponibilità dei servizi. Il primo load balancer, dedicato al front end dell'applicazione, gestisce in modo efficiente e bilanciato il traffico diretto agli utenti finali, distribuendo le richieste su due istanze operative per assicurare una risposta rapida e continua. Il secondo load balancer è stato configurato per gestire il backend dell'applicazione, che funge da servizio API principale. Questo load balancer gestisce le richieste provenienti da diverse fonti di dati e sistemi di elaborazione, garantendo una gestione robusta e ridondante delle operazioni. L'architettura del cluster Kubernetes supporta una scalabilità efficiente e una

crescita fluida delle capacità operative del sistema, mantenendo elevati standard di prestazioni e resilienza anche durante picchi di utilizzo e carichi di lavoro intensi.

3.4 Azure Cosmos DB for Mongo DB

Azure Cosmos DB for MongoDB è una soluzione robusta per utilizzare MongoDB all'interno dell'ecosistema Azure, offrendo scalabilità globale, compatibilità con MongoDB, modelli di dati flessibili e garanzie di servizio avanzate. Azure Cosmos DB è stato utilizzato per ospitare un database che contiene le statistiche dei giocatori della NBA. Questo implica che Azure Cosmos DB è stato configurato per memorizzare, gestire e rendere disponibili i dati delle prestazioni dei giocatori, inclusi punti, rimbalzi, assist, e altre metriche importanti per l'analisi.

Capitolo 4

Scalabilità, Affidabilità e Sicurezza

L'architettura del nostro applicativo è stata progettata con particolare attenzione a tre aspetti fondamentali: affidabilità, scalabilità e sicurezza delle connessioni.

L'affidabilità è garantita attraverso una strategia di duplicazione dei pod all'interno di Kubernetes.

La scalabilità, sia orizzontale che verticale, è implementata mediante Azure Kubernetes Service (AKS), infine la sicurezza delle connessioni è assicurata dall'uso di HTTPS.

4.1 Scalabilità automatica in Azure

La scalabilità è un aspetto fondamentale dell'architettura del nostro progetto, progettato per garantire che il sistema possa gestire un numero crescente di richieste mantenendo prestazioni ottimali sotto diverse condizioni di carico. La nostra infrastruttura, per quanto riguarda la scalabilità orizzontale si basa su Azure Kubernetes Service (AKS), integrando un node pool con due nodi che supportano l'auto-scaling. Per la scalabilità verticale, questa viene invece fornita dallo Scaler KEDA. Questo scaler si occupa di controllare quando la risorsa Backend si trova in una situazione di sforzo, andando ad aumentare le sue risorse di conseguenza.

Scalabilità Orizzontale

La scalabilità orizzontale, nota anche come scaling out, consiste nell'aggiunta di ulteriori nodi al cluster Kubernetes quando la domanda aumenta. Con l'auto-scaling configurato, Azure Kubernetes Service può automaticamente aggiungere nodi al node pool per gestire il carico di lavoro aggiuntivo. Questo è particolarmente vantaggioso in scenari di traffico variabile o durante picchi di utilizzo, permettendo all'applicazione di mantenere elevate prestazioni senza interventi manuali.

Nel contesto del nostro progetto, questa funzionalità è cruciale. I pod che eseguono i servizi front end, sviluppati in React, e i servizi back end, sviluppati in Flask, possono essere distribuiti su nuovi nodi in risposta a un aumento della domanda. Il load balancer distribuisce il traffico tra

i nodi disponibili, assicurando un bilanciamento efficace del carico e prevenendo il sovraccarico di singoli nodi.

Scalabilità verticale

La scalabilità verticale, o *scaling up*, si riferisce all'aumento delle risorse (CPU, memoria) dei nodi esistenti. Questa forma di scalabilità è necessaria quando le risorse attuali dei nodi non sono sufficienti per gestire i carichi di lavoro. Azure Kubernetes Service consente di ridimensionare verticalmente i nodi nel cluster, migliorando le capacità di calcolo e di memoria dei nodi esistenti.

Per il nostro progetto, la scalabilità verticale è particolarmente utile per gestire operazioni intensive che richiedono maggiori risorse per singoli pod. Ad esempio, il servizio backend che utilizza Flask e librerie di intelligenza artificiale come TensorFlow e Keras può beneficiare dell'aumento delle risorse per fornire tempi di risposta rapidi e gestire elaborazioni complesse.

In conclusione l'architettura del nostro progetto è stata quindi progettata per essere altamente scalabile, con la capacità di adattarsi automaticamente alle variazioni del carico di lavoro attraverso l'auto-scaling dei nodi e il ridimensionamento verticale delle risorse. Questa configurazione assicura che l'applicazione possa mantenere alte prestazioni e affidabilità anche sotto condizioni di carico variabile, garantendo un'esperienza utente ottimale e una gestione efficiente delle risorse.

4.2 Repliche dei Pod

L'affidabilità del servizio è un elemento chiave della nostra architettura, progettato per garantire la disponibilità continua e la resilienza dell'applicazione. Per migliorare l'affidabilità, la nostra infrastruttura utilizza Azure Kubernetes Service (AKS) con una configurazione di pod duplicati per i servizi front end e back end. Questa strategia di replicazione assicura che ci siano sempre istanze ridondanti dei pod disponibili. In caso di malfunzionamento o aggiornamenti di un pod, altri pod duplicati possono continuare a servire le richieste senza interruzioni, garantendo un'esperienza utente ininterrotta.

In sintesi, l'architettura del nostro servizio è progettata per garantire un'elevata affidabilità attraverso la duplicazione dei pod di Kubernetes, assicurando la continuità del servizio e la disponibilità dei dati anche in caso di guasti o manutenzione.

4.3 HTTPS

Per utilizzare HTTPS è stato necessario implementare TLS sia nel frontend che nel backend. Per quanto riguarda il frontend React è stato generato un file `.env` per gestire le variabili

d'ambiente:

```
HTTPS=true
SSL_CERT_FILE=nba-frontend.crt
SSL_KEY_FILE=nba-frontend.key
PORT=443
```

- **HTTPS=true:** Indica che l'applicazione React deve essere eseguita utilizzando HTTPS anzichè HTTP.
- **SSL_CERT_FILE=nba-frontend.crt:** Questa variabile specifica il nome del file certificato SSL (Secure Sockets Layer) per l'applicazione. Il certificato SSL è un file crittografico che viene utilizzato per stabilire l'autenticità del server e crittografare la comunicazione tra il client (browser) e il server.
- **SSL_KEY_FILE=nba-frontend.key:** Questa variabile specifica il nome del file chiave SSL associato al certificato. La chiave SSL è necessaria per decriptare i dati crittografati durante la comunicazione sicura.
- **PORT=443:** Questa variabile specifica la porta su cui l'applicazione React è configurata per ascoltare le richieste HTTPS. La porta 443 è la porta predefinita per le connessioni HTTPS.

In questo modo viene generato l'environment react con la connessione HTTPS e i certificati autofirmati.

E' stato inoltre configurato il Dockerfile del contenitore del frontend in modo che esponesse la connessione HTTPS con il seguente comando:

```
CMD [ "serve", "-s", "build", "-l", "443", "-ssl-cert",  
      "nba-frontend.crt", "-ssl-key", "nba-frontend.key" ]
```

Infine nel file di configurazione del deployment di Kubernetes è stato specificato di utilizzare per ogni load balancer la porta 443 in modo da garantire la connessione sicura con HTTPS.

metadata:

```
name: scalableprojectnba-frontend-service
```

spec:

```
type: LoadBalancer
```

```
ports:
```

```
- port: 443
```

```
  targetPort: 443
```

```
selector:
```

```
app: scalableprojectnba-frontend
```

Per quanto riguarda l'API, le azioni per garantire HTTPS sono analoghe a quelle per il frontend, vengono infatti generati i due file per il certificato e la chiave della connessione, dopo di che vengono utilizzati questi file sia nel container Docker che nel deployment di Kubernetes in modo tale da poter offrire l'API tramite richieste HTTPS.

4.4 Test

Per valutare le performance dell'applicazione in uno scenario realistico, è stato eseguito un test di carico utilizzando il framework locust. A questo scopo sono stati eseguiti due script python separati, uno per il frontend e l'altro per il backend, di seguito i due script:

- Frontend test script:

```
from locust import HttpUser, task, between
class WebsiteUser(HttpUser):
    wait_time = between(1, 5)
    def on_start(self):
        self.client.verify = False
    @task
    def load_frontend(self):
        self.client.get("/")
```

-Backend test script:

```
from locust import HttpUser, task, between
import random
team_names = [...]
class WebsiteUser(HttpUser):
    wait_time = between(1, 5)
    def on_start(self):
        self.client.verify = False
    @task(3)
    def access_api(self):
        headers = {"X-API-KEY": "taylor"}
        # Selezione random di due squadre
        selected_teams = random.sample(team_names, 2)
        params = {"squadra1": selected_teams[0],
                  "squadra2": selected_teams[1]}
        self.client.get("/api/predict", headers=headers, params=params)
```

Il framework Locust permette di accedere ad una GUI dedicata all'indirizzo "http://localhost:8089", dalla quale è possibile configurare:

- Il nome dell'host verso il quale fare le richieste
- Il numero massimo di user simulati da creare
- Il ramp-up, ovvero quanti utenti simulati vengono generati al secondo

Il valore di host è stato impostato, rispettivamente, agli indirizzi IP di frontend e backend; il numero di utenti a 100 e il ramp-up a 10 utenti al secondo.

I risultati del test sono soddisfacenti, tenendo conto delle risorse limitate utilizzabili da Azure. Di seguito, il testo di carico sul frontend:




Figura 4: Test di carico frontend - grafico

Si osserva un incremento costante nel tempo rispetto al numero di richieste al secondo. Questo indica che il sistema è stato in grado di gestire un numero crescente di richieste senza mostrare segni di cedimento o rallentamenti significativi. Questo è un indicatore positivo di scalabilità sotto carico.

Il tempo medio di risposta scende in una fase iniziale per poi stabilizzarsi a un valore molto basso e costante. Questo suggerisce che, nonostante l'aumento delle richieste, il sistema è in

grado di mantenere tempi di risposta rapidi e consistenti. Inoltre, il fatto che il 95° percentile segua una traiettoria simile alla media indica che la maggior parte delle richieste viene elaborata in tempi comparabili, con poche eccezioni che si discostano significativamente dalla norma.

Locust

HOST

https://172.213.194.22

STATUS

STOPPED

RPS


32.5

FAILURES

0%

NEW

RESET



STATISTICS

CHARTS

FAILURES

EXCEPTIONS

CURRENT RATIO

DOWNLOAD DATA

LOGS

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	1683	2	16	100	120	22.5	7	140	654.22	32.5	0
	Aggregated	1683	2	16	100	120	22.5	7	140	654.22	32.5	0

Figura 5: Test di carico frontend - logs

Come si può notare dalla figura 3, su circa 1700 richieste sono stati registrati solamente 2 fallimenti.

Il carico sul backend è stato valutato con modalità analoghe a quello impiegato per il frontend. Tuttavia, considerando la maggiore frequenza di richieste verso un'API rispetto agli accessi a un sito web, il test sul backend è stato condotto aumentando il numero di richieste per utente.

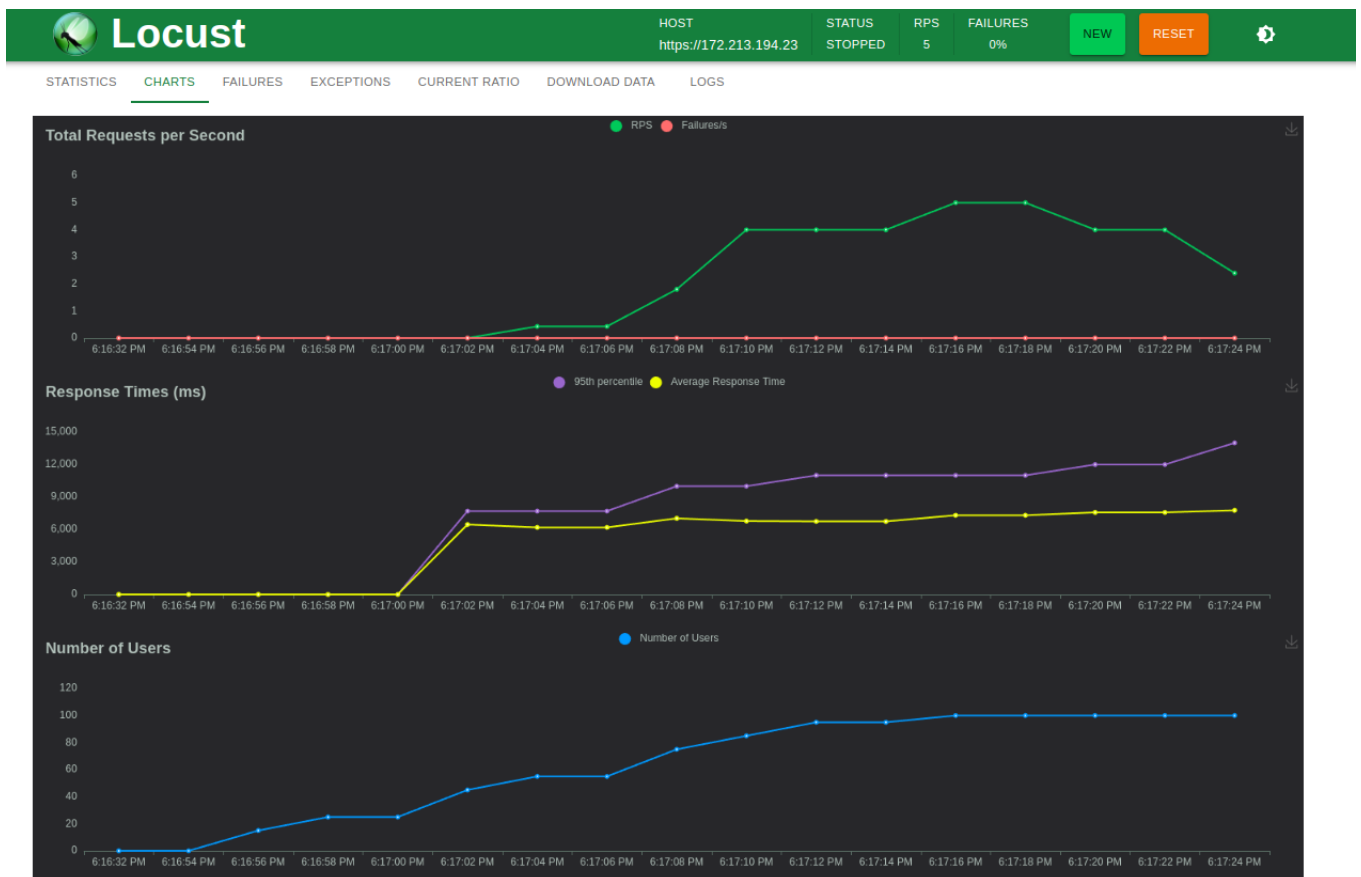


Figura 6: Test di carico backend - chart

Il grafico mostra un incremento costante e uniforme del numero totale di richieste al secondo. Inizia con un numero relativamente basso di richieste, che cresce fino a stabilizzarsi a un plateau. All'aumentare delle richieste, il tempo di risposta rimane stabile, indicando che il backend è in grado di gestire un numero crescente di richieste senza soffrire di significative degradazioni delle prestazioni. La capacità di scalare in modo lineare con l'incremento del carico è una dimostrazione di robustezza.

I tempi di risposta sono stabili su valori ragionevoli, tenendo conto che ogni richiesta comporta il calcolo della predizione della rete neurale.

Capitolo 5

Sviluppi futuri e conclusioni

Gli sviluppi futuri per la nostra applicazione includono diverse implementazioni chiave. Innanzitutto, la possibilità di integrare nodi aggiuntivi per migliorare la scalabilità e la distribuzione del carico di lavoro del sistema. Questo permetterà di gestire un numero maggiore di richieste e di migliorare le performance complessive dell'applicazione.

Inoltre, implementare una pipeline CI/CD utilizzando Azure DevOps. Questa pipeline automatizzerà il processo di build, test e deployment, garantendo un rilascio continuo e affidabile delle nuove funzionalità e delle correzioni di bug, riducendo al minimo i tempi di inattività e migliorando l'efficienza del team di sviluppo.

Infine, aggiungere la funzionalità di mostrare nel frontend lo scoreboard live delle partite della giornata futura. Questo offrirà agli utenti un'esperienza più coinvolgente e informativa, permettendo loro di predire i risultati di partite reali.

Il progetto svolto si è rivelato sfidante e formativo. Il team ha avuto l'opportunità di sperimentare l'utilizzo di nuove tecnologie e di approcciarsi allo sviluppo in modo innovativo, affrontando nuove difficoltà legate al mondo infrastrutturale e operativo, nonché alla vasta gamma di servizi Azure disponibili.

Il coordinamento e la suddivisione del lavoro tra i membri del team hanno giocato un ruolo fondamentale nello sviluppo del progetto, contribuendo in modo significativo all'ottimizzazione dei tempi di realizzazione.