Relazione sul Progetto "Videogame Store"

Nome: Antonio Binanti

Matricola: 1000053846

Introduzione

Il progetto "Videogame Store" è stato sviluppato con l'obiettivo di creare una piattaforma di e-commerce dedicata alla vendita di videogiochi, in grado di offrire agli utenti un'esperienza di acquisto semplice e veloce. L'applicazione è distribuita attraverso micro-servizi, utilizzando tecnologie moderne come Docker, Kubernetes, Prometheus per il monitoring, e Kafka per la comunicazione asincrona tra i micro-servizi.

Architettura dell'applicazione

L'architettura dell'applicazione è progettata come un sistema distribuito basato su micro-servizi, ognuno dei quali è responsabile di un insieme specifico di funzionalità. Questo approccio consente una gestione flessibile, scalabile e resiliente, in cui i singoli servizi sono indipendenti e comunicano tra loro tramite API HTTP/REST e un sistema di messaggistica asincrona tramite Kafka. I principali componenti di questa architettura sono:

Frontend

Il frontend rappresenta l'interfaccia utente dell'applicazione, consentendo agli utenti di interagire con il sistema. Esso invia richieste al backend per ottenere o inviare dati, utilizzando il protocollo HTTP tramite un API Gateway che funge da intermediario. Il frontend si occupa della gestione della logica dell'interfaccia utente, inclusi l'autenticazione, la visualizzazione dei giochi, degli ordini, delle notifiche e l'interazione con altre funzionalità dell'applicazione.

API Gateway (nginx)

L'API Gateway svolge un ruolo fondamentale nel reindirizzare le richieste degli utenti ai vari micro-servizi di backend. Questo è implementato utilizzando NGINX, configurato come reverse proxy. L'API Gateway offre un punto di ingresso centralizzato, che migliora la sicurezza, l'affidabilità e la gestione del traffico. Le richieste vengono inoltrate a ciascun servizio backend (Game Catalog, Order, Notification, etc.), a seconda della tipologia di operazione richiesta.

Micro-servizi di Backend

I micro-servizi costituiscono il cuore dell'architettura, ognuno dei quali gestisce una funzionalità specifica dell'applicazione. Ogni servizio espone un'insieme di API HTTP che consente al frontend di interagire con le sue risorse. I micro-servizi principali sono:

- **Game Catalog Service**: Gestisce il catalogo dei videogiochi, incluse le operazioni di aggiunta, aggiornamento, rimozione dei giochi e la gestione delle recensioni. Questo servizio si integra con un database MongoDB per memorizzare i dati relativi ai giochi e alle recensioni degli utenti.

- **Order Service**: Si occupa della gestione degli ordini degli utenti, inclusi acquisti, prenotazioni e dettagli sugli ordini. Utilizza un database PostgreSQL per mantenere le informazioni relative agli utenti e agli ordini effettuati.
- **Notification Service**: Gestisce le notifiche per gli utenti. Le notifiche possono riguardare nuovi ordini, nuovi giochi preferiti, o messaggi generali. Redis è utilizzato come sistema di caching per le notifiche, consentendo un recupero rapido e la gestione dello stato di lettura delle notifiche.

Kafka (Sistema di Messaggistica Asincrona)

Kafka è utilizzato per facilitare la comunicazione asincrona tra i micro-servizi. Permette ai servizi di inviare e ricevere messaggi in modo scalabile e resiliente, migliorando l'affidabilità e la gestione delle operazioni distribuite. Ad esempio, quando un ordine viene effettuato, il microservizio **Order Service** può inviare un messaggio tramite Kafka che il **Notification Service** utilizzerà per inviare notifiche all'utente.

Prometheus e Grafana (Monitoraggio e Analisi delle Performance)

La gestione e il monitoraggio delle performance dei vari micro-servizi è realizzata tramite Prometheus, un sistema di monitoraggio open-source, e Grafana, una piattaforma di visualizzazione dei dati. Prometheus raccoglie metriche relative alle performance dei servizi, come il numero di richieste HTTP, la durata delle richieste e le metriche del database, mentre Grafana viene utilizzato per visualizzare queste metriche attraverso dashboard personalizzabili.

- **Prometheus** raccoglie metriche relative alle performance, come il numero di richieste HTTP, la durata delle richieste, il carico del database, e altre informazioni cruciali per la diagnosi dei problemi.
- **Grafana** fornisce una visualizzazione interattiva dei dati di monitoraggio, permettendo agli amministratori di sistema di ottenere una panoramica delle performance in tempo reale.

Kubernetes (Orchestrazione e Auto-scalabilità)

Kubernetes è utilizzato per gestire l'orchestrazione e il deploy dei container. Kubernetes gestisce il ciclo di vita dei container, come il provisioning, il bilanciamento del carico e l'auto-scalabilità, permettendo una gestione automatica dei micro-servizi.

Database e Caching

Ogni micro-servizio si integra con i suoi rispettivi sistemi di gestione dei dati:

- **MongoDB**: Utilizzato dal **Game Catalog Service** per la gestione dei dati relativi ai giochi e alle recensioni.

- **PostgreSQL**: Utilizzato dall'**Order Service** per la gestione degli ordini e delle prenotazioni degli utenti.
- **Redis**: Usato dal **Notification Service** per memorizzare temporaneamente lo stato delle notifiche e migliorare le performance del servizio di notifica.

Servizi di Infrastruttura Aggiuntivi

Oltre ai micro-servizi principali, l'infrastruttura comprende anche alcuni servizi aggiuntivi che supportano l'operatività dell'applicazione, tra cui:

- **Zookeeper**: Gestisce la configurazione di Kafka e facilita la gestione della sua architettura distribuita.
- **Cadvisor**: Utilizzato per monitorare l'utilizzo delle risorse dei container, fornendo metriche sul consumo di CPU, memoria, e I/O.
- **Predictor**: Utilizza un modello ARIMA per fare previsioni sulle risorse (come CPU e richieste HTTP) in base ai dati storici, aiutando a ottimizzare l'allocazione delle risorse nei vari servizi.

Sicurezza e Autenticazione

L'autenticazione degli utenti è gestita attraverso token JWT (JSON Web Token). I vari micro-servizi implementano la logica di autenticazione e autorizzazione, garantendo che solo gli utenti autenticati possano accedere alle informazioni sensibili come gli ordini o le recensioni. L'API Gateway di NGINX è configurato per proteggere l'accesso alle API esposte dai micro-servizi.

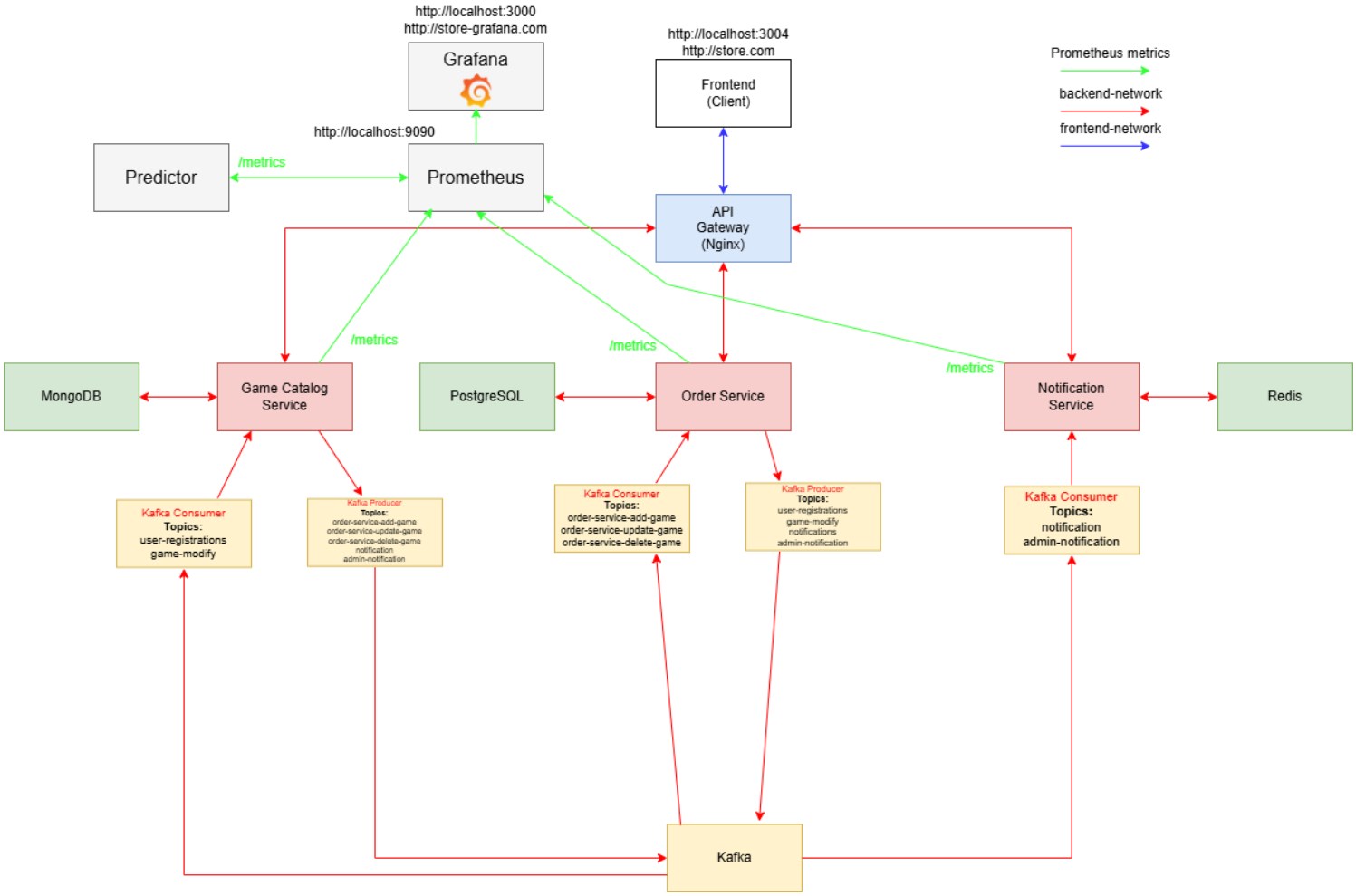
Scalabilità e Affidabilità

Grazie all'uso di **Kubernetes**, l'architettura può facilmente scalare orizzontalmente (aggiungendo più istanze di un servizio) e verticalmente (aumentando le risorse di una singola istanza). Kubernetes si occupa di gestire l'auto-scalabilità in base al carico, mentre i micro-servizi possono essere ridistribuiti in modo trasparente per garantire alte performance anche in caso di picchi di traffico.

Diagramma dell'Architettura

Il diagramma architetturale mostra i micro-servizi distribuiti su Docker o Kubernetes, con il frontend che comunica tramite il reverse proxy (NGINX) con i servizi di backend, Kafka come

sistema di messaggistica asincrona collega i vari micro-servizi, mentre Prometheus e Grafana raccolgono e visualizzano le metriche di monitoraggio.



Micro-servizi e API Implementate

L'architettura dei micro-servizi è stata progettata per ottimizzare la separazione delle responsabilità e la scalabilità. Ogni servizio è indipendente, con una chiara responsabilità e un set specifico di API per interagire con il resto del sistema.

Game Catalog Service

Questo servizio si occupa della gestione del catalogo dei videogiochi e delle recensioni. È progettato per consentire agli utenti di cercare, aggiungere, aggiornare o rimuovere giochi e di interagire con le recensioni.

File principali:

- **main.py**: Gestisce il routing e le chiamate API.

- **metrics.py**: Esporta le metriche per il monitoraggio con Prometheus.
- **kafka_producer.py e kafka_consumer.py**: Implementano la logica di comunicazione asincrona con Kafka.
- **mongo_db.py**: Contiene la logica di interazione con il database MongoDB.

API esposte:

- **/metrics**: Esposizione delle metriche per Prometheus, come il numero di richieste, latenza media e utilizzo delle risorse.
- **/getUserPreferredGames**: Recupera i giochi preferiti dall'utente, basandosi su dati storici o preferenze impostate.
- **/getGameByTitle/<string:title>**: Restituisce le informazioni di un gioco specifico tramite il titolo.
- **/addGame**: Consente di aggiungere un nuovo gioco al catalogo. Richiede informazioni come titolo, genere, prezzo, e descrizione.
- **/updateGame**: Permette l'aggiornamento dei dettagli di un gioco già esistente nel catalogo.
- **/deleteGame/<string:title>**: Rimuove un gioco specifico dal catalogo tramite il titolo.
- **/addReview**: Aggiunge una recensione per un gioco. Richiede i dettagli come username, titolo del gioco e contenuto della recensione.
- **/getReviewByGame/<string:title>**: Restituisce tutte le recensioni associate a un determinato gioco.

Integrazione con Kafka:

- Gli eventi come l'aggiunta di un nuovo gioco o una recensione vengono pubblicati su Kafka per essere consumati da altri servizi, come il Notification Service.

Order Service

Il servizio di gestione ordini è responsabile delle funzionalità di registrazione e login degli utenti, oltre a gestire prenotazioni e acquisti.

File principali:

- **main.py**: Gestisce il routing e le API principali.
- **metrics.py**: Esporta le metriche per Prometheus.
- **kafka_producer.py e kafka_consumer.py**: Implementano la comunicazione asincrona con Kafka per notifiche e sincronizzazioni.

- **db_postgres.py**: Contiene le operazioni di lettura e scrittura sul database PostgreSQL.

API esposte:

- **/metrics**: Esposizione delle metriche di sistema, come il numero di ordini processati o la latenza delle API.
- **/signup**: Registra un nuovo utente nel sistema. Richiede username, password e dettagli opzionali come email.
- **/login**: Consente agli utenti di autenticarsi. Restituisce un token JWT per l'accesso sicuro.
- **/getUser/<string:username>**: Restituisce i dettagli di un utente, come informazioni personali e storico degli ordini.
- **/getReservations/<string:username>**: Recupera tutte le prenotazioni di un utente specifico.
- **/addReservation**: Aggiunge una nuova prenotazione. Richiede dettagli come username, titolo del gioco e data.
- **/deleteReservation/<string:reservation_id>**: Rimuove una prenotazione dal sistema in base all'ID.
- **/addPurchase**: Registra un acquisto effettuato da un utente. Invia un evento Kafka per generare notifiche.
- **/getPurchases/<string:username>**: Restituisce l'elenco di tutti gli acquisti di un utente.

Integrazione con Kafka:

- Quando viene aggiunto un ordine, viene generato un messaggio Kafka che può essere consumato dal Notification Service per informare l'utente.

Notification Service

Questo servizio gestisce tutte le notifiche per gli utenti, come notifiche di acquisto, nuove recensioni o l'inserimento di nuovi giochi nel catalogo. Utilizza Redis per la gestione rapida dei dati e Kafka per la ricezione di eventi dai servizi correlati.

File principali:

- **main.py**: Gestisce le chiamate API per la gestione delle notifiche.
- **metrics.py**: Esporta le metriche relative al numero di notifiche elaborate e al tempo di risposta.
- **kafka_consumer.py**: Consuma eventi Kafka relativi a nuovi ordini o aggiornamenti.

- **db_redis.py**: Gestisce l'interazione con Redis, utilizzando chiavi per ogni utente e notifiche non lette.

API esposte:

- **/metrics**: Fornisce le metriche del servizio per Prometheus, come il numero di notifiche inviate.
- **/getAllNotifications/<string:username>**: Recupera tutte le notifiche di un determinato utente.
- **/getUnreadNotifications/<string:username>**: Restituisce le notifiche non lette di un utente specifico.
- **/markNotificationAsRead**: Permette di segnare una notifica come letta. Richiede l'ID della notifica.

Gestione di Redis:

- Redis viene utilizzato per mantenere le notifiche non lette e quelle lette, offrendo tempi di accesso rapidi e una scalabilità adeguata.

Integrazione con Kafka:

- Consuma eventi generati dai servizi Game Catalog e Order per notificare l'utente su nuovi giochi o acquisti.

Frontend

Il frontend gestisce tutte le interazioni utente, fungendo da punto di ingresso principale dell'applicazione. È responsabile della navigazione, della presentazione dei dati e della gestione delle notifiche in tempo reale.

File principali:

- **File di interfaccia utente**: Implementano la logica di visualizzazione e gestione dei componenti interattivi.
- **Routing Flask**: Utilizzato per gestire le chiamate verso le API del backend.
- **Chiamate API**: Implementano le richieste alle API dei micro-servizi per il recupero di dati come giochi, recensioni e notifiche.

Funzionalità principali:

- **Home Page**: Presenta i giochi disponibili, con opzioni per cercare, filtrare e ordinare.
- **Dashboard utente**: Consente di gestire notifiche, prenotazioni e acquisti.

- **Integrazione notifiche**: Visualizza le notifiche in tempo reale, recuperandole dal Notification Service.
- **Autenticazione e autorizzazione**: Basata su token JWT, con gestione sicura delle sessioni.

Questa struttura di micro-servizi garantisce modularità, scalabilità e facilità di manutenzione, permettendo all'applicazione di evolvere in modo agile e di rispondere alle esigenze degli utenti in modo efficiente.

Modalità di Distribuzione e Deploy

L'applicazione è progettata per essere distribuita facilmente utilizzando Docker Compose o Kubernetes. Entrambi gli approcci garantiscono una configurazione semplificata, l'orchestrazione dei micro-servizi e un'esperienza di distribuzione uniforme.

Distribuzione con Docker Compose

Docker Compose consente di avviare rapidamente tutti i micro-servizi e i relativi componenti (come database e broker di messaggi) tramite un singolo comando.

Passaggi:

1. **Pre-requisiti:**
    - Assicurarsi che **Docker** e **Docker Compose** siano installati nel sistema.
    - Verificare l'integrità del file docker-compose.yml presente nella directory principale del progetto.
2. **Comando per la build e l'avvio dei servizi:** Utilizzare il seguente comando per costruire e avviare tutti i container:

```
docker-compose up --build
```

3. **Tempi di avvio:**
    - Attendere circa **1-2 minuti** affinché tutti i container siano completamente operativi.
4. **Accesso all'applicazione:**
    - Una volta avviati i servizi, è possibile accedere all'applicazione tramite un browser all'indirizzo:

    ```
    http://localhost:3004/
    ```

5. **Gestione dei container:**
    - Per fermare i servizi, utilizzare:

```
docker-compose down
```

    - Per visualizzare i log:

```
docker-compose logs -f
```

Distribuzione con Kubernetes

Kubernetes offre un'architettura più complessa ma altamente scalabile e adatta a sistemi di produzione.

Passaggi:

1. **Installazione di Ingress NGINX:**
    - Kubernetes utilizza un controller Ingress per la gestione del traffico HTTP verso i servizi. Per configurare il controller NGINX, eseguire:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/cloud/deploy.yaml
```

    - Questa operazione crea le risorse necessarie per gestire il traffico ingress.
2. **Modifica del file /etc/hosts:**
    - Per consentire al browser di risolvere i nomi host configurati, aggiungere le seguenti righe al file /etc/hosts:

```
127.0.0.1 store.com
```

```
127.0.0.1 store-grafana.com
```

    NB: Questa modifica richiede privilegi amministrativi.
3. **Creazione delle immagini Docker:**
    - Ogni micro-servizio richiede la creazione di un'immagine Docker prima del deploy. Per fare ciò:
        - Posizionarsi nella directory contenente il file Dockerfile del servizio.
        - Eseguire il comando:

```
docker build -t <nome_servizio>:1.0 .
```

    - Ripetere l'operazione per i seguenti servizi:
        - **frontend**

        - **game-catalog**
        - **order-service**
        - **notification-service**
        - **predictor**

4. **Deploy con Kubernetes:**
    - Applicare la configurazione dei servizi utilizzando il file k8s.yml:

    ```
    kubectl apply -f k8s.yml
    ```

    - Questo comando creerà i pod, i servizi, e le risorse necessarie per il funzionamento dell'applicazione.

5. **Gestione dei pod e dei servizi:**
    - Verificare lo stato dei pod con:

    ```
    kubectl get pods
    ```

    - In caso di problemi, riavviare un pod specifico:

    ```
    kubectl delete pod <nome_pod>
    ```

Kubernetes ricreerà automaticamente il pod.

6. **Accesso all'applicazione:**
    - Una volta avviati i servizi, accedere all'applicazione tramite il browser utilizzando:

    ```
    http://store.com/login
    ```

Accesso in Modalità Admin

L'applicazione include una modalità amministratore per gestire funzionalità avanzate. Le credenziali di accesso predefinite sono:

- **Username:** admin
- **Password:** 0000

Questa modalità consente di accedere a funzionalità amministrative, come la gestione avanzata del catalogo, il monitoraggio delle metriche e la gestione degli utenti.

Monitoring

L'infrastruttura di monitoring è stata progettata per coprire due aspetti fondamentali:

1. **Monitoring White-Box**: Consente di monitorare le performance delle richieste ai servizi e dei processi interni, fornendo una visione dettagliata sul comportamento delle applicazioni.
2. **Monitoring Black-Box**: Si concentra sul monitoraggio delle risorse computazionali utilizzate dagli ambienti di esecuzione, incluse CPU, memoria, I/O del file system e rete.

Monitoring White-Box

Per implementare un sistema di monitoring white-box più completo e adattato alle esigenze del progetto, è stata integrata una soluzione personalizzata per la raccolta delle metriche direttamente dalle richieste del backend. La strategia utilizzata include l'instrumentazione del codice applicativo tramite **Prometheus** e un sistema di **annotation** per monitorare le richieste HTTP e altre metriche significative.

Nei microservizi di backend, ogni richiesta viene monitorata grazie a funzioni di pre- e post-elaborazione che registrano metriche chiave come la durata della richiesta e il tipo di risposta.

E' stata adottata la soluzione basata su **Prometheus con annotation custom** per le seguenti ragioni:

1. **Semplicità e Robustezza**: Prometheus è già parte del sistema, ed estendere il monitoring per raccogliere metriche delle richieste HTTP è stato naturale e non ha richiesto l'introduzione di nuovi componenti.
2. **Efficienza**: La raccolta e la registrazione diretta delle metriche durante l'elaborazione delle richieste riduce il rischio di perdita di dati ed elimina la necessità di pipeline complesse.
3. **Scalabilità**: Prometheus supporta grandi volumi di dati e offre integrazione diretta con Grafana per la visualizzazione.
4. **Flessibilità**: Le metriche raccolte possono essere facilmente estese o aggregate con query PromQL, consentendo analisi avanzate senza modificare l'applicazione.

Strumenti utilizzati:

- **Prometheus**: Raccoglie e archivia le metriche esposte dai servizi tramite endpoint specifici.

- **Grafana**: Fornisce una visualizzazione grafica delle metriche raccolte, tramite dashboard personalizzabili.

Passaggi per la configurazione:

1. Accesso a Grafana

- **Docker Compose**: Accedere tramite browser all'indirizzo:

  http://localhost:3005/

- **Kubernetes**: Accedere all'indirizzo:

  http://store-grafana.com

- Credenziali predefinite:
  - **Username:** admin
  - **Password:** admin

2. Configurazione di Prometheus come Data Source

1. Accedere a Grafana e navigare su:

   Home > Connections > Data Sources > Add data source

2. Selezionare **Prometheus** come tipo di data source.
3. Configurare l'URL di Prometheus:
   - Per Docker Compose:

     http://prometheus:9090

   - Per Kubernetes:

     http://prometheus.default.svc.cluster.local:9090

4. Cliccare su **Save & Test** per verificare la connessione.

3. Creazione di una Dashboard su Grafana

1. Dalla barra laterale, selezionare:

   > Dashboard

2. Cliccare su **Add new panel** per configurare un pannello.
3. Selezionare **Prometheus** come data source.
4. Nella barra delle query, aggiungere le metriche da monitorare, ad esempio:
   - **http_requests_total**: Per il conteggio totale delle richieste HTTP.
   - **http_request_duration_seconds**: Per monitorare la durata delle richieste HTTP.

- **db_requests_total**: Per il numero totale di richieste al database.
- **db_request_duration_seconds**: Per la durata delle richieste al database.
- **kafka_messages_processed_total**: Per monitorare i messaggi elaborati da Kafka.

5. Personalizzare il pannello con grafici, valori o tabelle e cliccare su **Apply** per salvare.

4. Utilizzo diretto di Prometheus

1. Accedere a Prometheus tramite browser:
   - **Docker Compose**:

   http://localhost:9090
2. Nella sezione **Graph**, digitare le query Prometheus per visualizzare le metriche. Ad esempio:
   - **http_requests_total**
   - **http_request_duration_seconds**
3. Cliccare su **Execute** per eseguire la query e visualizzare i risultati in formato grafico o tabellare.

Monitoring Black-Box

Il monitoring black-box è realizzato tramite **cAdvisor**, un tool integrato per il monitoraggio delle risorse computazionali a livello di container.

Configurazione e accesso:

- I dati di **cAdvisor** sono accessibili tramite Grafana, previa configurazione come data source.
- Le metriche raccolte da cAdvisor sono utili per monitorare l'utilizzo delle risorse, come CPU, memoria, rete e I/O.

Metriche principali di cAdvisor:

- **CPU**:
  - **container_cpu_usage_seconds_total**: Tempo totale di utilizzo della CPU.
  - **container_cpu_load_average_1m**: Media di carico della CPU nell'ultimo minuto.
- **Memoria**:

    - **container_memory_usage_bytes**: Utilizzo della memoria da parte del container.

- **File System**:
    - **container_fs_reads_bytes_total**: Bytes letti dal file system.
    - **container_fs_writes_bytes_total**: Bytes scritti sul file system.
    - **container_fs_usage_bytes**: Utilizzo totale del file system.
    - **container_fs_limit_bytes**: Limite di utilizzo del file system.
- **Rete**:
    - **container_network_receive_bytes_total**: Bytes ricevuti tramite la rete.
    - **container_network_transmit_bytes_total**: Bytes trasmessi tramite la rete.
- **I/O del disco**:
    - **container_blkio_sync_total**: Operazioni di I/O sincrone.
    - **container_blkio_async_total**: Operazioni di I/O asincrone.
    - **container_blkio_sectors_total**: Settori letti o scritti sul disco.
- **Eventi**:
    - **container_last_event_timestamp**: Timestamp dell'ultimo evento significativo.

Visualizzazione delle metriche:

Utilizzare Grafana per creare dashboard che combinino metriche white-box e black-box.

Predittore ARIMA

L'**ARIMA** (AutoRegressive Integrated Moving Average) è un modello di previsione statistico che analizza i dati storici e identifica le tendenze temporali. Nel contesto del predittore, è stato configurato per effettuare previsioni su:

- **Utilizzo della CPU** (predicted_cpu_usage)
- **Numero di richieste HTTP per servizio** (predicted_http_requests)

Le previsioni sono basate su un'analisi delle metriche raccolte negli ultimi 5 minuti (finestra temporale di 5m).

Architettura del Predittore

Il servizio è implementato in Python, utilizzando:

- **Flask**: Per esporre le API.
- **Prometheus Client**: Per pubblicare le metriche previste.
- **Statsmodels**: Per il calcolo delle previsioni tramite il modello ARIMA.

File Principale: predictor.py

Di seguito le principali funzionalità del file predictor.py:

1. **Fetch delle Metriche**:
    - Il predittore esegue richieste a Prometheus per ottenere i dati storici delle seguenti metriche:
        - http_requests_total[5m]
        - container_cpu_usage_seconds_total[5m]
    - I dati recuperati vengono elaborati e convertiti in un DataFrame Pandas.
2. **Applicazione del Modello ARIMA**:
    - Per ciascun job individuato nei dati raccolti, viene applicato un modello ARIMA configurato con i parametri (1, 1, 1).
    - Il modello genera una previsione per i prossimi 10 intervalli temporali, con una frequenza di 15 secondi.
3. **Esposizione delle Metriche Predette**:
    - Le previsioni generate vengono pubblicate su Prometheus tramite due metriche:
        - **predicted_cpu_usage**: Utilizzo previsto della CPU, con label timestamp e job.
        - **predicted_http_requests**: Richieste HTTP previste, con label timestamp e job.
4. **Endpoint Esposti**:
    - **/metrics**: Espone le metriche aggiornate per Prometheus.
    - **/predict**: Esegue manualmente il processo di predizione.
    - **/**: Endpoint di health check per verificare lo stato del servizio.

Configurazione in Grafana

Le previsioni sono integrate nella dashboard di **Grafana** per una visualizzazione intuitiva.

Accesso e Configurazione:

1. Accedere a Grafana tramite:
    - **Docker Compose**: http://localhost:3005/
    - **Kubernetes**: http://store-grafana.com
2. Configurare Prometheus come data source (vedi sezione Monitoring White-Box).

Visualizzazione delle Previsioni:

- **Metriche di Previsione**:
    - **predicted_cpu_usage**: Mostra l'utilizzo della CPU previsto.
    - **predicted_http_requests**: Mostra il numero di richieste HTTP previsto per ogni servizio.
- **Filtri Label**:
    - Per filtrare le previsioni relative a un servizio specifico, utilizzare il filtro di label exported_job.
- **Suggerimento di Visualizzazione**:
    - Per una rappresentazione chiara dei dati, utilizzare il tipo di visualizzazione **Bar Gauge**:
        - Navigare su **Visualizations** in alto a destra nella dashboard.
        - Selezionare **Bar Gauge**.

Metriche Esposte

Le principali metriche previste e i loro utilizzi sono:

1. **predicted_cpu_usage**:
    - Predizione dell'utilizzo della CPU per ogni servizio (job).
    - Label:
        - **timestamp**: Momento della previsione.
        - **job**: Servizio monitorato.
2. **predicted_http_requests**:
    - Predizione del numero di richieste HTTP per i servizi.
    - Label:

- **timestamp**: Momento della previsione.
- **job**: Servizio monitorato.

Funzionalità del Predittore

1. Raccolta Dati:
I dati vengono recuperati tramite query Prometheus. Ad esempio:

- Query per richieste HTTP:

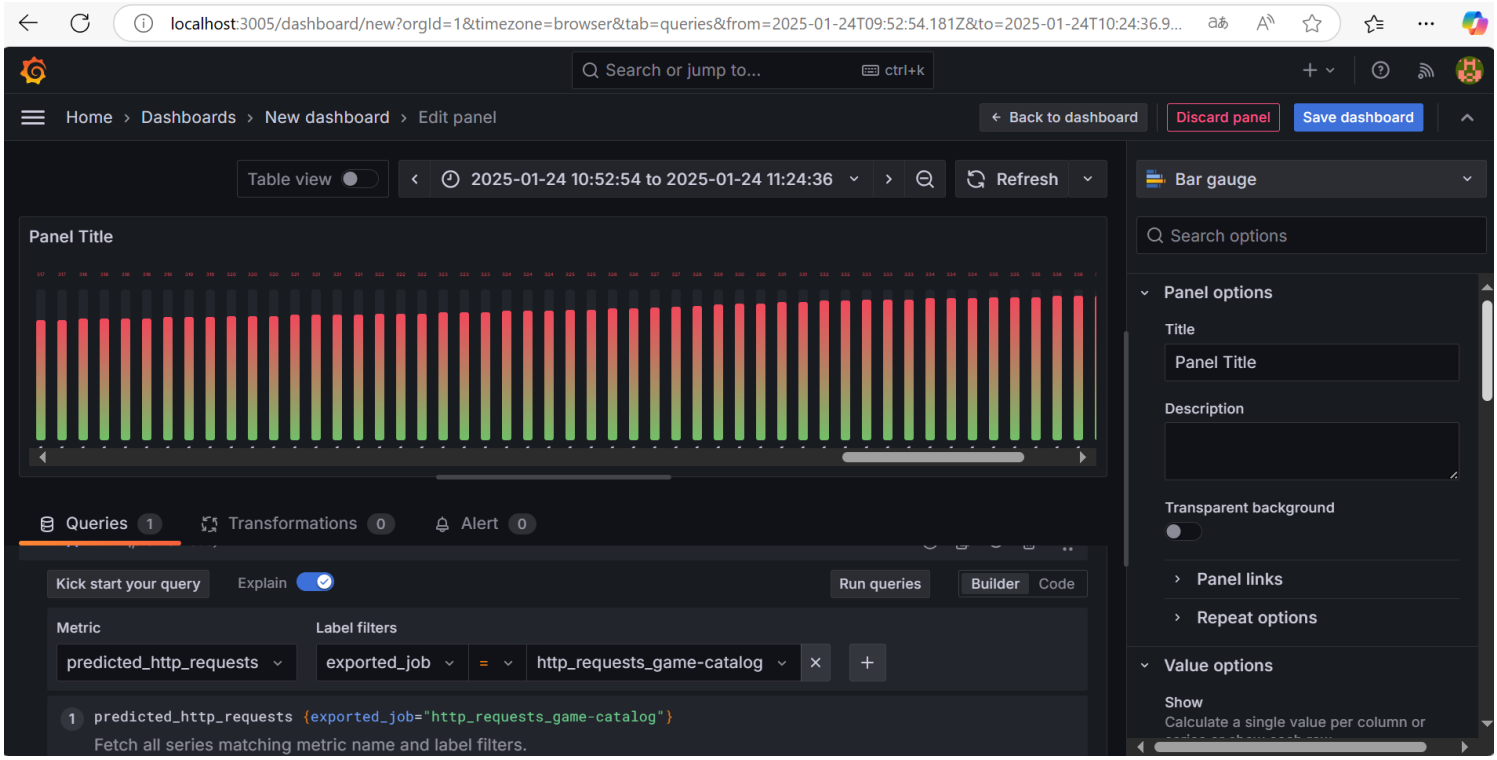
  `http_requests_total[5m]`

- Query per utilizzo CPU:

  `container_cpu_usage_seconds_total[5m]`

2. Previsione con ARIMA:
Ogni dataset viene elaborato tramite il modello ARIMA, che genera previsioni per i prossimi 10 intervalli temporali.

3. Esposizione Dati:
Le previsioni vengono pubblicate su Prometheus e sono pronte per essere visualizzate in Grafana.



Conclusioni

Il progetto "Videogame Store" è una piattaforma robusta che offre funzionalità avanzate di e-commerce e gestione delle risorse. La scelta di un'architettura basata su micro-servizi, con l'uso di tecnologie moderne come Docker, Kubernetes, Kafka, Prometheus e Grafana, assicura scalabilità, monitoraggio avanzato e un'ottima esperienza utente.