Progetto DSBD

Rino Di Paola 1000027309
Gianluca Grasso 1000027121

https://github.com/gianlucagrassog/insurance-microservices-k8s-docker

1. Descrizione dell'applicazione

Il progetto consiste nella realizzazione di un software per la gestione di una compagnia assicurativa. I microservizi implementati sono quattro: uno per la gestione degli utenti (*user-service*), uno per la gestione delle polizze assicurative (*policy-service*), uno per la gestione della sottoscrizione alle assicurazioni (*purchase-sercice*) e uno per la generazione delle ricevute e l'invio delle stesse (*receipt-service*). Ogni microservizio (ad eccezione di receipt-service) è connesso a un database MongoDB. Un utente può sottoscrivere un'assicurazione alla volta, selezionando una polizza (ad esempio: bonus malus, temporanea, a chilometri, ecc.) e una lista di garanzie accessorie desiderate (ad esempio: furto e incendio, infortuni del conducente, kasko, ecc.), che se selezionate comportano un aumento del prezzo dell'assicurazione. Il microservizio Policy si occupa quindi di calcolare il prezzo dell'assicurazione, basandosi sul tipo di polizza assicurativa scelta, sull'anagrafica dell'utente e sulle garanzie accessorie richieste dallo stesso. Ogni microservizio espone delle API Rest, che consentono di effettuare tutte le operazioni necessarie al funzionamento del sistema. Tre microservizi sono stati sviluppati in Java utilizzando il framework SpringBoot, mentre un microservizio (receipt-serivice) è stato sviluppato in Python. La consistenza dei dati è stata gestita tramite il modello di progettazione di Saga. La comunicazione è stata gestita tramite message broker Kafka. Inoltre, è stato applicato il pattern API Gateway, per gestire le API e aggregare i vari servizi richiesti. Per la gestione dei flussi asincroni di dati ed eventi è stata utilizzata la Reactive programming. Infine, è stato utilizzato Prometheus con approccio whitebox monitoring: in particolare, vengono monitorate le metriche relative all'applicazione (tentativi di registrazione di un utente, tentativi di registrazione di una polizza, tentativi di sottoscrizione di una polizza assicurativa mediante un acquisto).

2. Distribuzione dell'applicazione

La struttura a microservizi che è stata utilizzata si adatta perfettamente al tipo di applicazione, in quanto in futuro le varie parti del sistema potrebbero evolvere in maniera indipendente; inoltre la scalabilità potrebbe rappresentare un punto critico ed in particolare ci potrebbe essere un carico che si concentra su una o un'altra parte del sistema in momenti diversi. La scelta di un'architettura a microservizi garantisce dunque una migliore flessibilità e migliori prestazioni, con un uso più accurato delle risorse.

2.1 Docker, Docker Compose

Il deployment dell'applicazione è stato effettuato all'interno di container Docker. Per ogni microservizio, la Docker image viene creata mediante un Dockerfile e con il comando *docker build*.

Tramite Docker Compose, è stato possibile creare un file .yaml in cui sono definiti i servizi e, con un singolo comando, creare, avviare e collegarsi ai container.

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

Esempio di *service* definito nel file *docker-compose.yaml*:

```
  purchase-service:
    <<: *common-settings
    container_name: purchase-service
    build: ./purchase-service
    environment:
      MONGODB_HOSTNAME: mongo
      MONGODB_PORT: 27017
      MONGO_INITDB_ROOT_USERNAME: *mongo-user
      MONGO_INITDB_ROOT_PASSWORD: *mongo-pass
      MONGO_INITDB_DATABASE: *mongo-db
      KAFKA_ADDRESS: kafkaserver:9092
      KAFKA_GROUP_ID: purchase-service-group
      KAFKA_TOPIC_1: purchase-user-topic
      KAFKA_TOPIC_2: purchase-policy-topic
      KAFKA_TOPIC_4: purchase-receipt-topic
      KAFKA_TOPIC_5: receipt-purchase-topic
    ports:
      - '8080'
```

2.2 Kubernetes

Per orchestrare e gestire tutte le risorse dei container da un singolo piano di controllo è stato utilizzato Kubernetes. Quest'ultimo offre la possibilità di distribuire i container in modo agevole e scalabile e di gestire al meglio i carichi di lavoro.

Gli oggetti Service, definiti nei file .yaml, sono delle astrazioni che consentono di esporre un'applicazione in esecuzione su un set di pod come un servizio di rete.
Esempio (purchase-service):

```
apiVersion: v1
kind: Service
metadata:
  name: purchase-service
spec:
  ports:
    - port: 8080
  selector:
    app: purchase-service
```

Gli oggetto Deployment, definiti nei file .yaml, consentono di gestire set di pod identici. Quest'oggetto ha il compito di creare i pod, assicurarsi che rimangano aggiornati e che ce ne siano abbastanza in esecuzione.
Esempio (purchase-service):

```
apiVersion: apps/v1
kind: Deployment
```

```yaml
metadata:
  name: purchase-service
spec:
  selector:
    matchLabels:
      app: purchase-service
  template:
    metadata:
      labels:
        app: purchase-service
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "8080"
        prometheus.io/path: "/actuator/prometheus"
    spec:
      containers:
        - name: purchase-service
          image: purchase-service:v1
          ports:
            - containerPort: 8080
          envFrom:
            - configMapRef:
                name: purchase-service-db-env-file
            - configMapRef:
                name: purchase-service-env-file
            - secretRef:
                name: db-secret-file
```

3. Pattern

3.1 API Gateway

- Docker Compose

Per accedere alle risorse dei microservizi da un dominio comune è stato utilizzato Nginx, un web server open source, che può essere usato come reverse proxy.

```yaml
nginx-proxy:
    image: jwilder/nginx-proxy
    ports:
      - "80:80"
    volumes:
      - /var/run/docker.sock:/tmp/docker.sock:ro
      - ./vhost.d:/etc/nginx/vhost.d
```

File di configurazione *insurance.local*:

```
location /users {
  proxy_pass http://user-service:8080/;
}
location /purchases {
  proxy_pass http://purchase-service:8080/;
}
location /policies {
proxy_pass http://policy-service:8080;
}
location /optionals {
  proxy_pass http://policy-service:8080;
}
```

- Kubernetes

Per gestire le API e aggregare i vari servizi è stato utilizzato Ingress. Un Ingress è una regola che definisce come un servizio, presente all'interno di un cluster, possa essere reso disponibile all'esterno del cluster.

Per abilitare NGINX Ingress controller, è necessario eseguire il seguente comando:

```
minikube addons enable ingress
```

Configurazione del file *5-ingress.yaml* (esempio: *purchase-service*):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: spring-ingress
  annotations:
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-methods: "PUT, GET, POST, OPTIONS,
DELETE"
    nginx.ingress.kubernetes.io/cors-allow-origin: "*"
    nginx.ingress.kubernetes.io/cors-allow-credentials: "true"
    nginx.ingress.kubernetes.io/cors-allow-headers: "Content-Type"
    nginx.ingress.kubernetes.io/rewrite-target: "/"
spec:
  rules:
  - host: insurance.app.loc
    http:
      paths:
      - path: /purchases
        pathType: ImplementationSpecific
        backend:
          service:
            name: purchase-service
            port:
              number: 8080
```

3.2 Saga

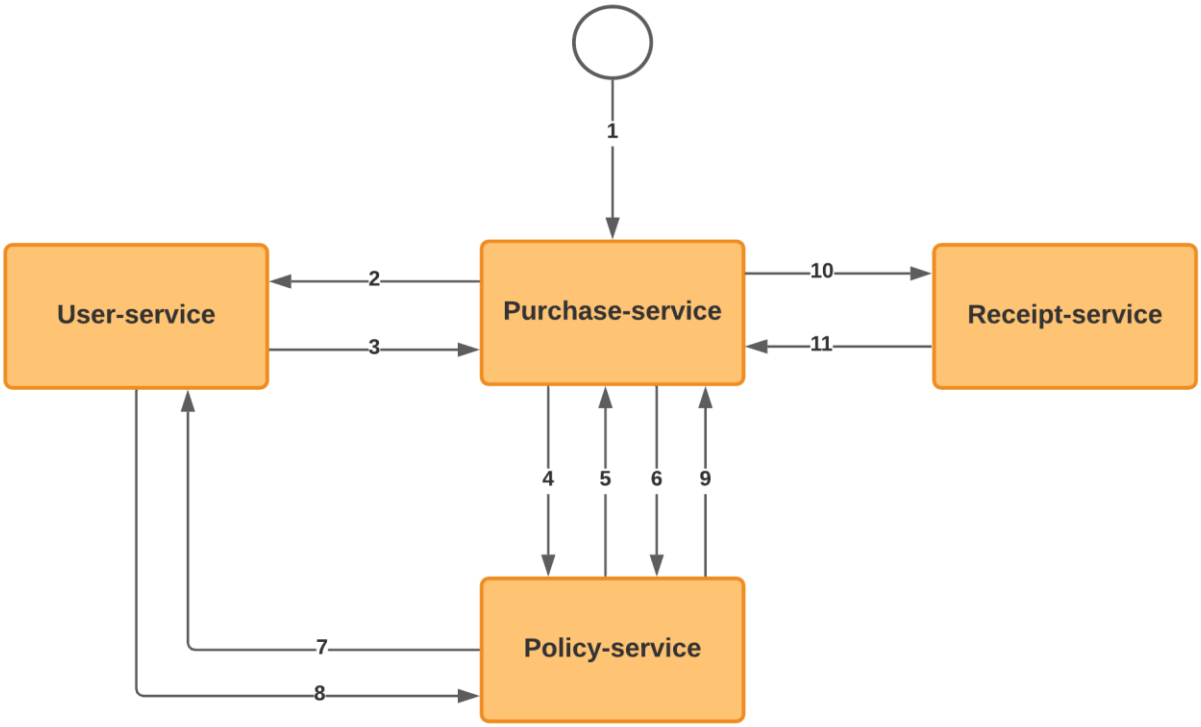
Si tratta di un pattern il cui scopo è garantire l'ordine di esecuzione delle transazioni che fanno parte della stessa saga, ovvero che devono essere eseguite in un ordine prestabilito e il cui effetto non è compromesso da eventuali transazioni intermedie appartenenti a saghe diverse. Questo pattern aiuta quindi a gestire la consistenza dei dati nell'esecuzione di transazioni distribuite tra microservizi diversi; coinvolge diversi attori (servizi) che vanno ad agire su una stessa entità tramite singole transazioni atte all'aggiornamento di un dato comune. L'obiettivo è duplice: mantenere l'identità del dato ed effettuare azioni di compensazione per ripristinarla in caso di errore.

Per questo progetto è stato utilizzato un approccio events/choreography: esso prevede che tutti i microservizi coinvolti lavorino innescati da eventi specifici, come in una vera e propria coreografia. Non esiste infatti, in questo approccio, un controllore che si occupi di amministrare tutto, ma ogni servizio sa esattamente cosa fare e quando farlo; l'unione, infatti, del "lavoro" di tutti i servizi sfocia nell'inizio, nell'avanzamento e nella conclusione della saga. Questo approccio è più complesso rispetto all'approccio di Saga basato su orchestrator. L'approccio di Saga basato su choreography è adatto a situazioni in cui si implementano nuovi microservizi da zero e quando a una singola transazione partecipa un numero limitato di microservizi (tutte caratteristiche presenti nella nostra applicazione).

Per i microservizi user, policy e purchase sono state implementate le seguenti API, basate su HTTP/REST:

- GET
  - "/": ottieni tutti gli elementi di una collezione.
  - "/{id}/exists": verifica se un elemento esiste nella collezione.
- POST
- - "/": crea un elemento nella collezione.
- DELETE
- - "/{id}": elimina un elemento dalla collezione.

Quando viene effettuata una richiesta POST al microservizio purchase, vengono eseguite delle transazioni che fanno parte della stessa saga. Di seguito la rappresentazione della saga:



1) Viene inviata una richiesta POST al microservizio purchase-service, con l'obiettivo di creare un nuovo acquisto. Vengono specificati: id dell'utente, id della policy, nome dell'ordine ed eventuali garanzie accessorie. Inizialmente viene generato un acquisto avente stato *PENDING*.

```
@PostMapping(path = "/", consumes = "application/JSON", produces =
"application/JSON")
    public Mono<Purchase> newPurchase(@RequestBody Purchase p) {
        metrics.increment();
        return repository.save(p).flatMap(purchase -> {
            kafkaTemplate.send(kafkaTopic1, "PolicyPurchase: Checking User|" +
p.get_user_string() + "|" + p.get_id_string());
            return Mono.just(p);
        });
    }
```

2) Tramite Kafka, purchase-service contatta user-service per verificare che l'id dell'utente esista. Oltre ad una stringa identificativa della richiesta, viene inviato sul topic l'id dell'utente che è stato specificato nella richiesta.

3) User-service, dopo aver ricevuto il messaggio, verifica l'esistenza dell'utente e pubblica sul topic una stringa in cui specifica se l'utente esiste o meno.

```
if (messageParts[0].equals("PolicyPurchase: Checking User")) {
            String uid = messageParts[1];
            repository.existsById(new ObjectId(uid)).flatMap(exists -> {
                kafkaTemplate.send(kafkaTopic1,
(exists?"UserExists|":"UserNotExists|") + messageParts[2]);
```

```
                return Mono.just(exists);
            }).subscribe();
        }
```

Se l'utente esiste, purchase-service setta lo stato dell'acquisto a *USER_CONFIRMED*, altrimenti a *REJECTED*.

4) Dopo aver verificato l'esistenza dell'utente, purchase-service contatta policy-service per verificare l'esistenza di una polizza avente l'id specificato nella richiesta.

5) Policy-service risponde a purchase-service, specificando se la polizza assicurativa esiste o meno.

6) In caso affermativo, purchase-service setta lo stato dell'acquisto a *POLICY_CONFIRMED*, altrimenti a *REJECTED*. Inoltre, se la polizza assicurativa esiste, purchase-service invia un messaggio di conferma a policy-service, in cui specifica anche l'id dell'acquisto, il nome dell'utente, il tipo e il prezzo della polizza e il prezzo degli optional.

7) Se policy-service riceve il messaggio di conferma da purchase-service, contatta user-service, in modo da ottenere le informazioni sull'anagrafica dello stesso, che saranno necessarie per il calcolo del prezzo dell'acquisto.

8) User-service risponde sullo stesso topic inviando, tra le altre cose, l'età e la classe di bonus malus.

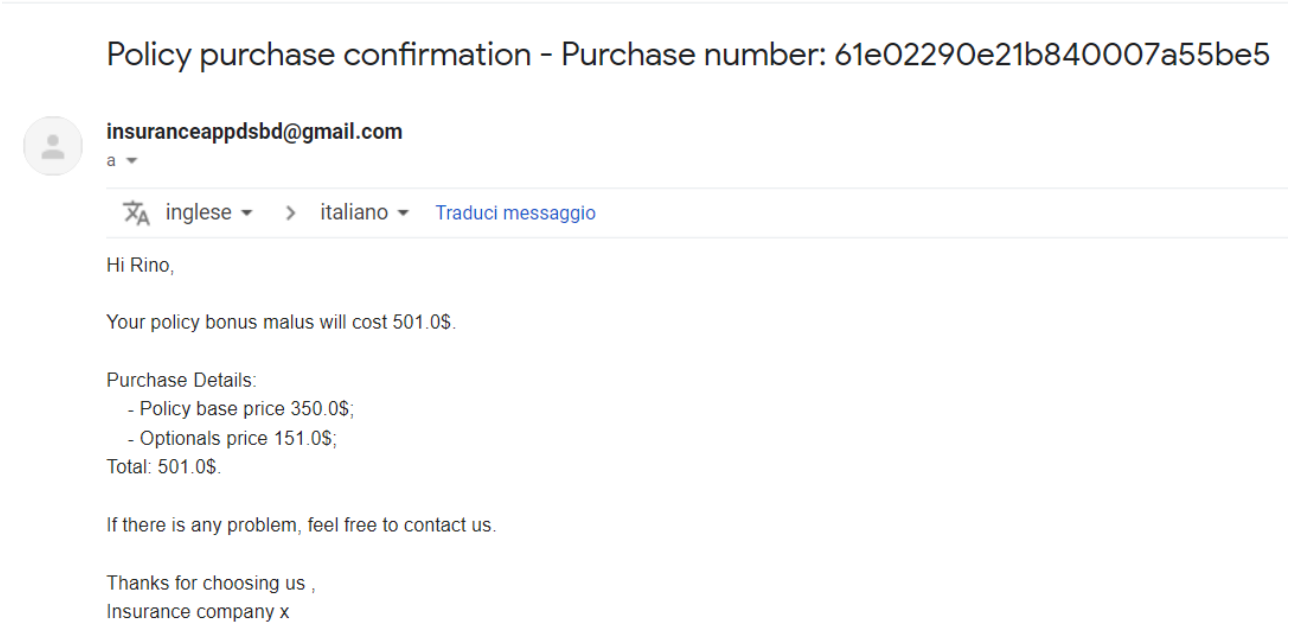
9) Una volta ricevute le informazioni dell'utente, policy-service calcola il prezzo totale dell'acquisto, sulla base dell'età dell'utente, del tipo di assicurazione (bonus malus, temporanea, ecc.) e del prezzo degli optional.

```
private double totalCalculator(double optionals_price, String policyType, int age,
int bmClass) {
    double totalPrice = optionals_price;
    if(age<23) {
        totalPrice += 100;
    }
    if(policyType.compareTo("bonus malus")==0) {
        totalPrice += (25 * bmClass);
    }
    else if(policyType.compareTo("temporanea")==0){
        totalPrice += 200;
    }
    return totalPrice;
}
```

Successivamente, invia tutte le informazioni a purchase-service.

10) Purchase-service setta lo stato dell'acquisto a PRICE_CALCULATED e contatta il microservizio receipt-service, specificando l'id dell'acquisto, il nome dell'utente, il tipo di polizza, il prezzo degli optional e il totale. Receipt-service, dopo aver ricevuto il messaggio, invia una mail all'indirizzo specificato contenente la ricevuta dell'acquisto.

Esempio di ricevuta:



Policy purchase confirmation - Purchase number: 61e02290e21b840007a55be5

insuranceappdsbd@gmail.com
a

inglese > italiano Traduci messaggio

Hi Rino,

Your policy bonus malus will cost 501.0$.

Purchase Details:
- Policy base price 350.0$;
- Optionals price 151.0$;

Total: 501.0$.

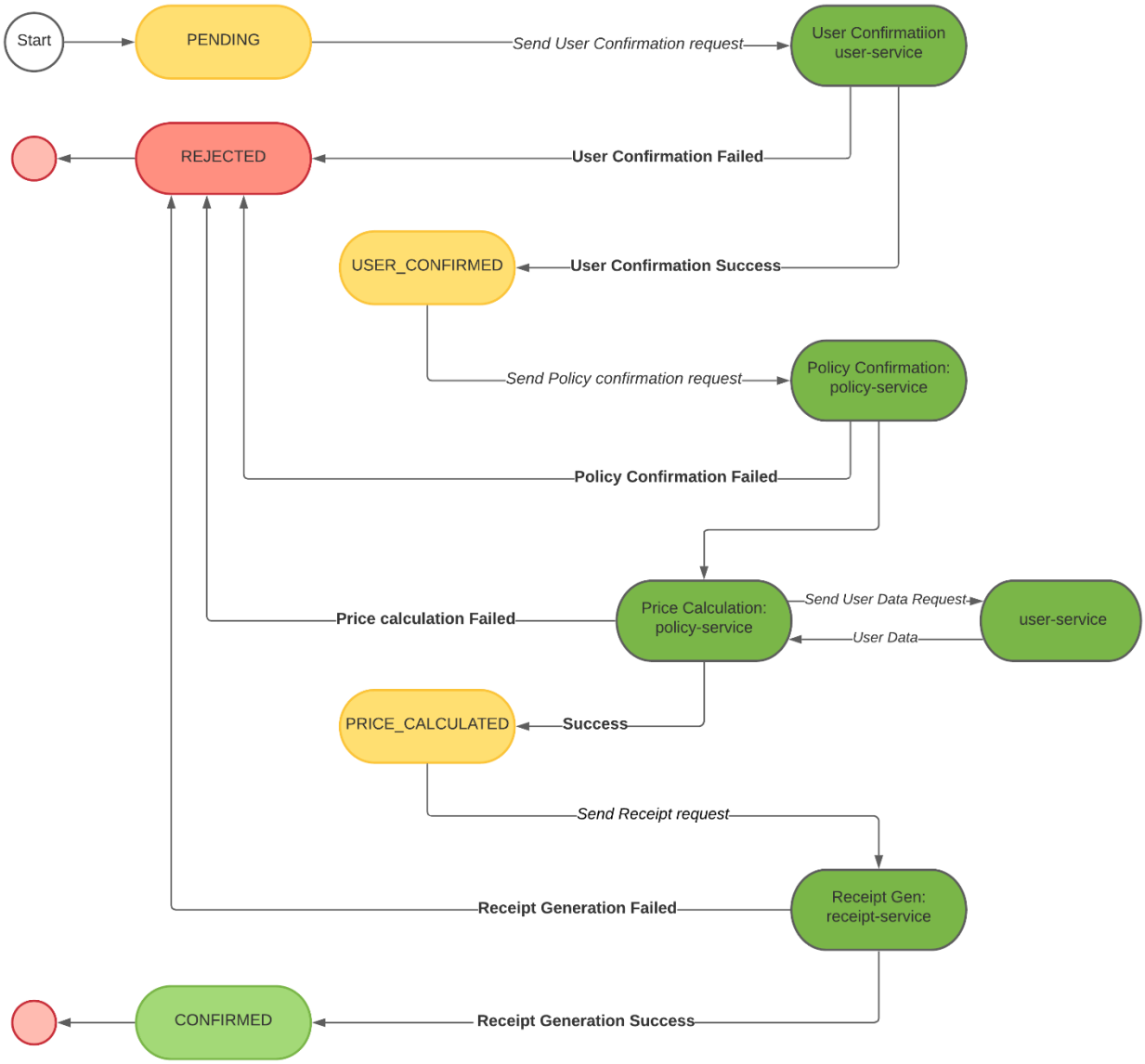
If there is any problem, feel free to contact us.

Thanks for choosing us ,
Insurance company x

11) Receipt-service, dopo aver inviato la mail, invia un messaggio di conferma a purchase-service.

12) Infine, purchase-service, dopo aver ricevuto un messaggio di conferma da receipt-service, setta lo stato dell'acquisto a CONFIRMED.

Macchina a stati SAGA:



4. White-box Monitoring

Le performance delle richieste sono state monitorate tramite un sistema di monitoring di tipo white-box. Quest'ultimo viene adottato in contesti in cui si ha piena visibilità dell'infrastruttura e dei servizi che caratterizzano l'applicazione e consente l'esportazione di informazioni sullo stato interno.
Per questo progetto la scelta del tipo di monitoraggio è ricaduta su white-box, in quanto la struttura interna, la progettazione e l'implementazione del programma sono note al tester.

4.1 Generazione delle metriche (Micrometer)

Micrometer è un set di librerie per Java, che funge da interfaccia tra l'applicazione e il tool di monitoraggio. Tramite Micrometer è possibile prelevare dei valori dall'applicazione e generare delle metriche, le quali potranno essere estratte da diversi tool (nel nostro caso da Prometheus).

Dipendenze e configurazione per Micrometer e Prometheus:

- File *pom.xml*:

```
<dependency>

    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-core</artifactId>
</dependency>

<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

- File *7-deployment.yaml*:

```
annotations:

    prometheus.io/scrape: "true"
    prometheus.io/port: "8080"
    prometheus.io/path: "/actuator/prometheus"
```

In questo progetto sono state generate tre differenti metriche, tutte di tipo counter: un contatore delle richieste POST al microservizio User (inserimento di un utente), un contatore delle richieste POST al microservizio Policy (inserimento di una polizza assicurativa) e un contatore delle richieste POST al microservizio Purchase (inserimento di un acquisto).

4.2 Simulazione dell'invio delle richieste

Come caso d'uso per questo progetto, sono state generate delle richieste POST ai tre microservizi tramite uno script Python. In particolare, si è fatto in modo di generare delle richieste in modo tale da ottenere delle time series con trend esponenziale.

```
x=random.exponential(scale=5, size=400)
x.sort()
noise = np.random.normal(0, 0.4, x.shape)
x = x + noise
x=x[:320]
max = np.amax(x)
for val in x:
    resp = requests.post(url, headers=headers, data=data)
    print(resp.status_code)
    time.sleep((max+0.5)-val)
```

[send_requests-purchase.py]

4.3 Generazione dei file .csv

Le metriche estratte da Prometheus tramite delle query nel linguaggio PromQL (in cui come parametro è stato specificato il range temporale di acquisizione, oltre che la metrica stessa) sono state salvate in dei file .csv. Quest'operazione di salvataggio può essere effettuata in due modi:

- Tramite Grafana, un'applicazione web per la visualizzazione e l'analisi interattiva di dati.
- Tramite una query in Python, utilizzando la libreria *prometheus_api_client*:

```
pu_ts = get_metric_value_in_time_range("purchase_counter_total", start_time,
end_time, chunk_size)
pu_ts = fix_ts(pu_ts)
f = open('purchase_metrics.csv','w+')
writer = csv.writer(f)
pu_ts.to_csv('purchase_metrics.csv')
```

[metric_exporter.py]

4.4 Analisi delle serie temporali (Python)

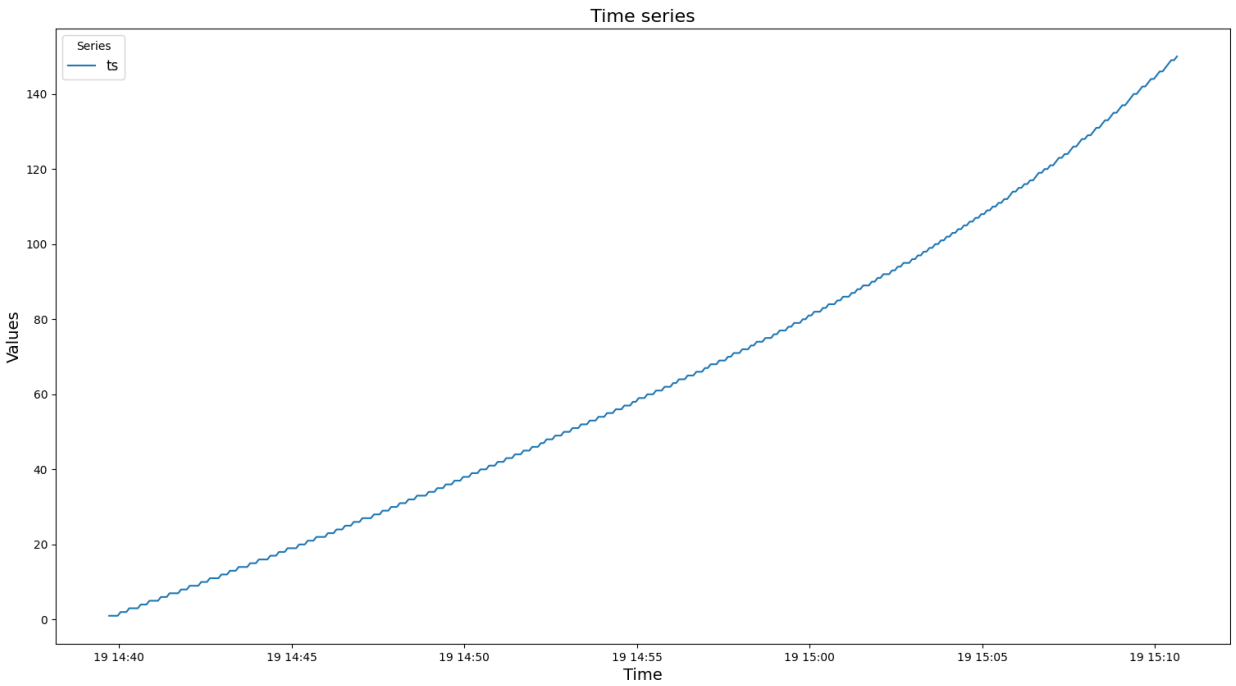
- userCounter [ts_analysis-user.py]

Il primo passo da effettuare è quello di importare i dati dal corrispondente file .csv e inserirli all'interno di un dataframe.

```
ts = pd.read_csv('user_metrics.csv', header=0, parse_dates=[0], index_col=[0])
```

Poiché tutti i dati provengono dalla stessa sorgente, non è necessario effettuare un'operazione di resampling.

E' possibile visualizzare la serie in un grafico utilizzando la libreria Matplotlib.

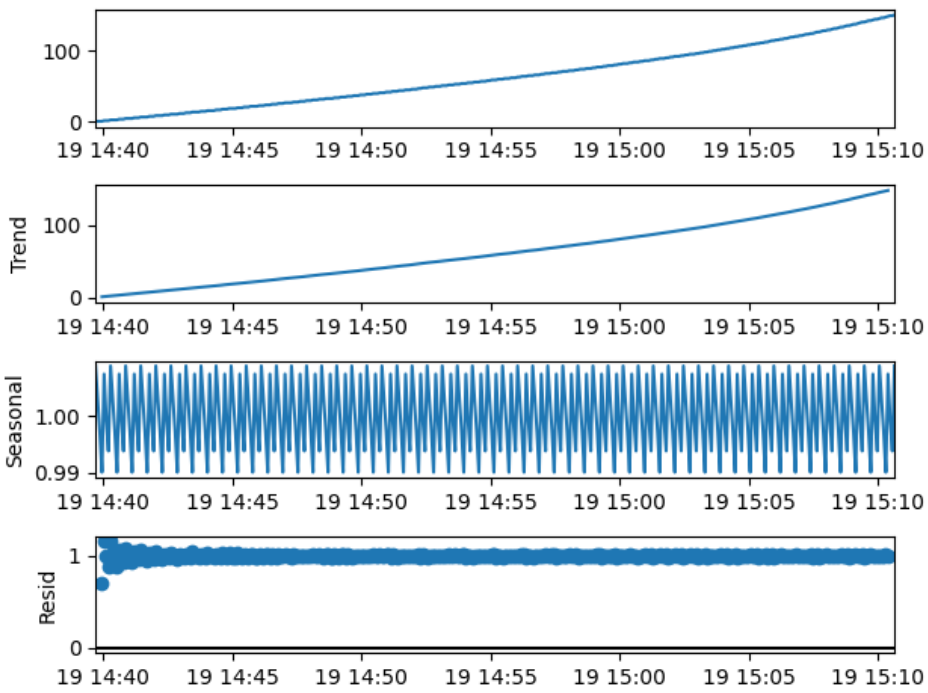


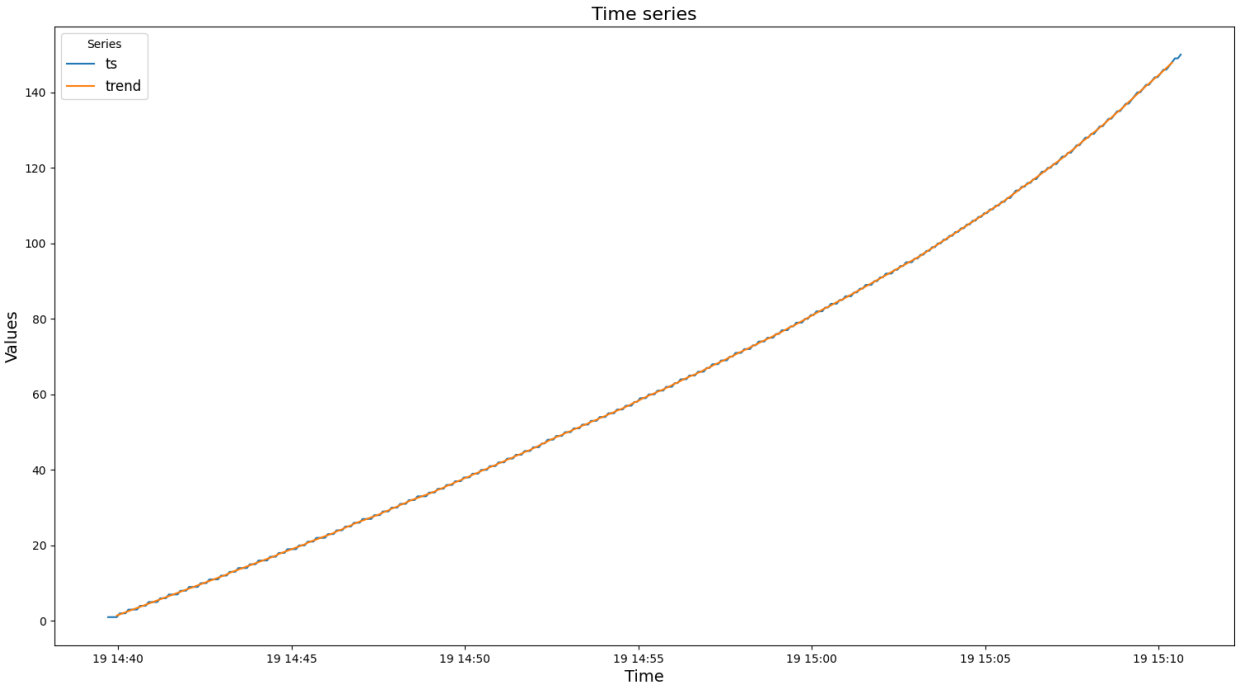
Analizzando il grafico, possiamo osservare un trend crescente, con andamento esponenziale.

Scomponiamo la serie tramite la funzione *seasonal_decompose* ed estraiamo il trend.

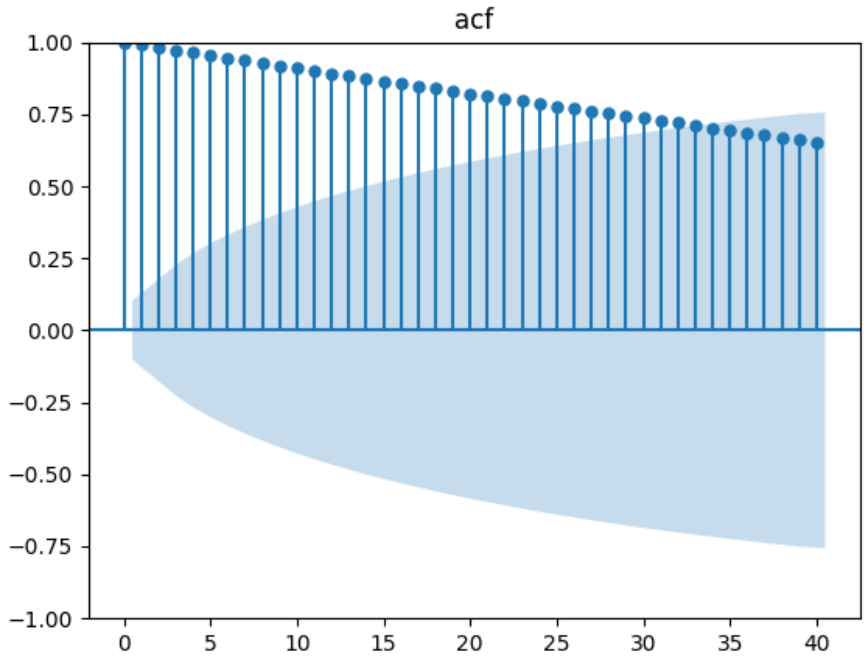
```
result = seasonal_decompose(ts, model='mul', period=7)
trend = result.trend.dropna()
```

La scelta dei parametri *model* e *period* va fatta sulla base dell'osservazione del grafico della serie. Generalmente, quando la serie ha un andamento esponenziale o logaritmico il parametro model va settato a '*mul*'. Fatto ciò, risulta opportuno tracciare un grafico del risultato e verificare che l'andamento del trend sia corretto e la media dell'errore sia circa uguale a 1 (o uguale a 0 nel caso in cui viene settato il model con il valore 'add').





La stagionalità può essere anche derivata dal grafico di autocorrelazione.



Poiché il grafico non presenta una forma sinusoidale, possiamo affermare che la serie non ha stagionalità.

Verifichiamo che la serie sia stazionaria, applicando il Dickey-Fuller test:

```
X = ts.values
adft = adfuller(X)
print('ADF Statistic: %f' % adft[0])
print('p-value: %f' % adft[1])
print('Critical Values:')
for key, value in adft[4].items():
    print('\t%s: %.3f' % (key, value))
```

Otteniamo i seguenti risultati:

```
ADF Statistic: 2.273665
p-value: 0.998938
Critical Values:
        1%: -3.449
        5%: -2.870
        10%: -2.571
ADF Statistic: 1.411880
p-value: 0.997167
```

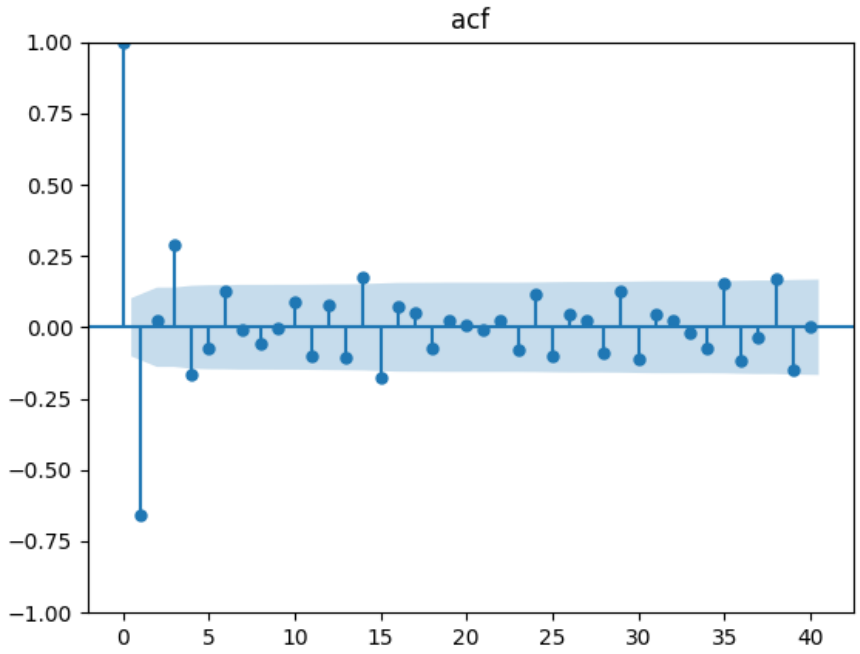
Poiché il valore ADF Statistic non è minore rispetto ai valori critici a differenti percentuali, non possiamo rifiutare le nostre null hypothesis; possiamo quindi concludere che la serie non è stazionaria. Per rendere stazionaria la serie, applichiamo un'operazione di differenza, che effettua la differenza tra un'osservazione e la precedente.

```
ts_diff = ts.diff()
```

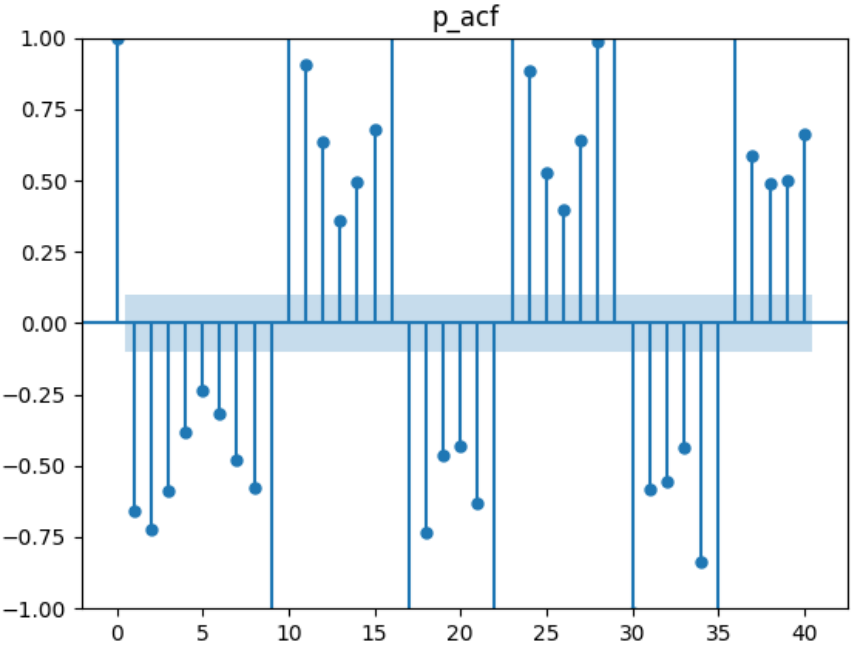
Applicando nuovamente il test di verifica di stazionarietà, si otterrà nuovamente che la serie non è stazionaria. Quindi, ripetiamo ancora una volta l'operazione.
Questa volta, dal test di stazionarietà otteniamo che la serie è stazionaria e quindi possiamo procedere con l'analisi.

Passi della predizione:

- Il modello scelto per l'analisi della nostra serie è ARIMA. La scelta di ARIMA piuttosto che SARIMA deriva dal fatto che la serie non presenta stagionalità. Per applicare il modello, è necessario ricavare i parametri p (ordine autoregressivo), d (ordine d'integrazione), q (ordine della media mobile).
p è ricavabile dal grafico do autocorrelazione:



In questo caso, p potrebbe essere 15.
Il valore di q può essere ricavato dal grafico di autocorrelazione parziale:



Possiamo affermare che q è uguale a 0.

Infine, d è uguale al numero di procedimenti effettuati per rendere la serie stazionaria e quindi è uguale a 2.

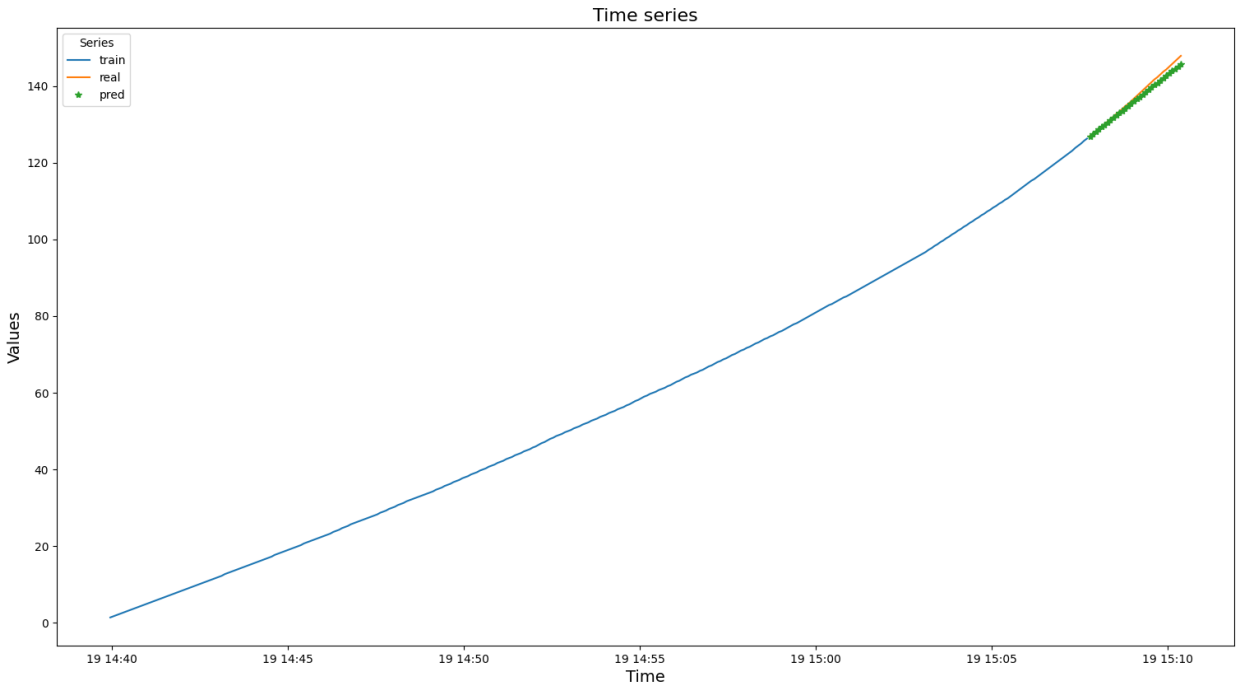
Un modo alternativo per calcolare i parametri p, d e q è quello di applicare la funzione *auto_arima*, che calcola l'order ottimale per un modello ARIMA.

- Splittiamo i dati in *train_data* e *test_data*, utilizzando il 90% dei campioni per train e il 10% per test.

```
train = trend.iloc[:334]
test = trend.iloc[334:]
```

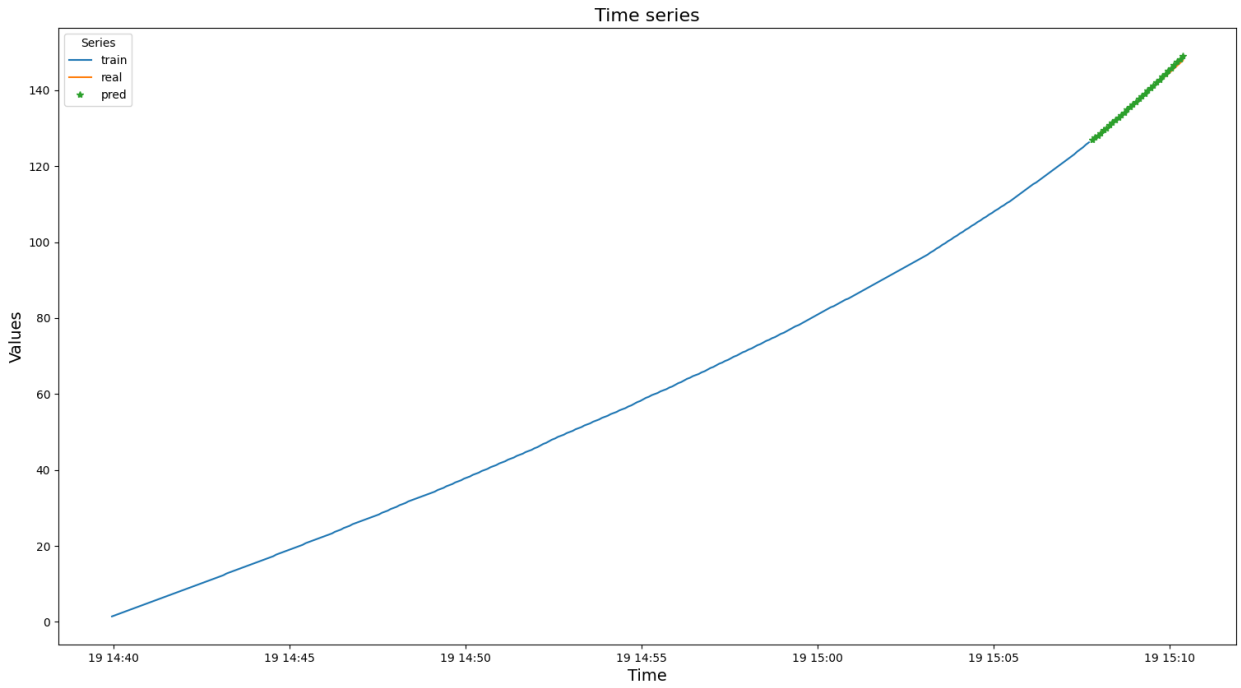
- Facciamo il fit del modello sul training set e valutiamolo sul test set.

```
model = ARIMA(train, order=(15,2,0))
results = model.fit()
start = len(train)
end = len(train)+len(test)-1
predictions = results.predict(start=start, end=end, dynamic=False, typ='levels')
```



Notiamo che questo modello matcha abbastanza bene la serie. Tuttavia, aumentando di un'unità il valore di d, otteniamo un modello (ARIMA con order=(15,3,0)) che matcha perfettamente la serie.

- Validazione del modello:
sovrapponendo il test sulla previsione effettuata, possiamo valutare l'efficacia del modello.



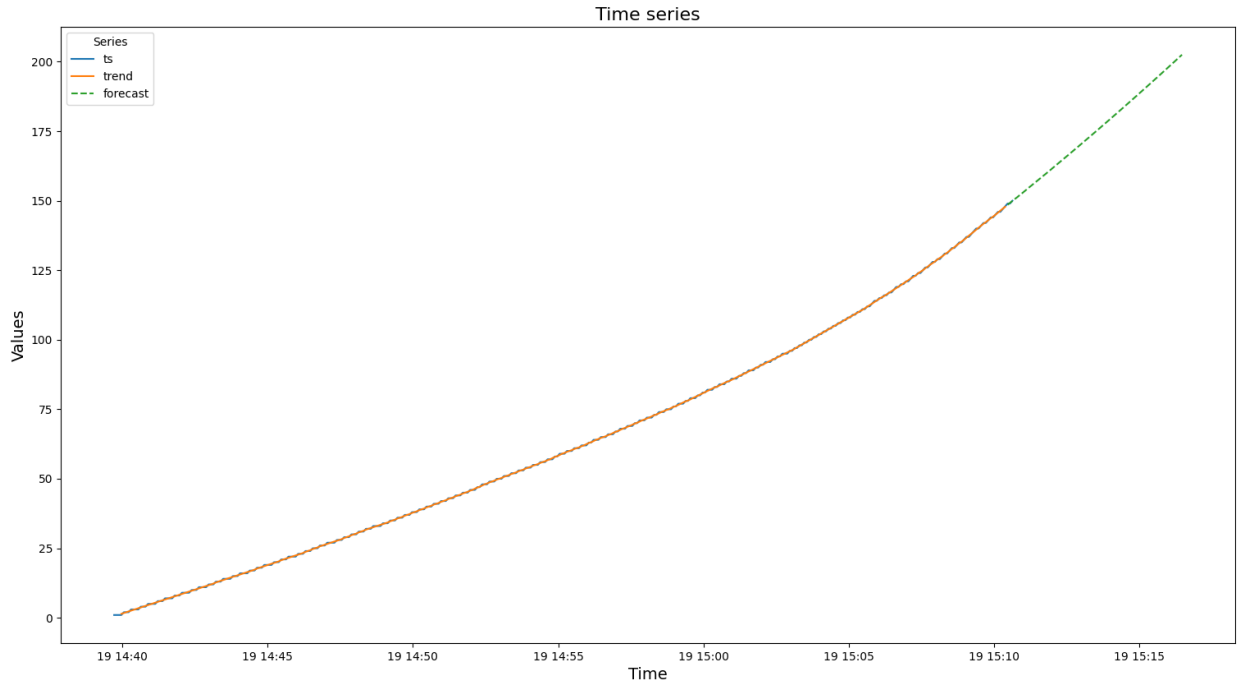
Calcolo dell'errore quadratico medio:
verifichiamo che l'errore che otteniamo sia inferiore rispetto alla standard deviation (std) della serie stessa.
E' possibile visualizzare la std della serie tramite la funzione *describe()*.

```
rmse(test, predictions)
mean_squared_error(test, predictions)
```

Più il valore degli errori che otteniamo è vicino a 0, più il modello si adatta a dati. Poiché gli errori risultano essere minori rispetto alla std, possiamo ritenere il modello valido e procedere con il suo utilizzo per predire i dati futuri.

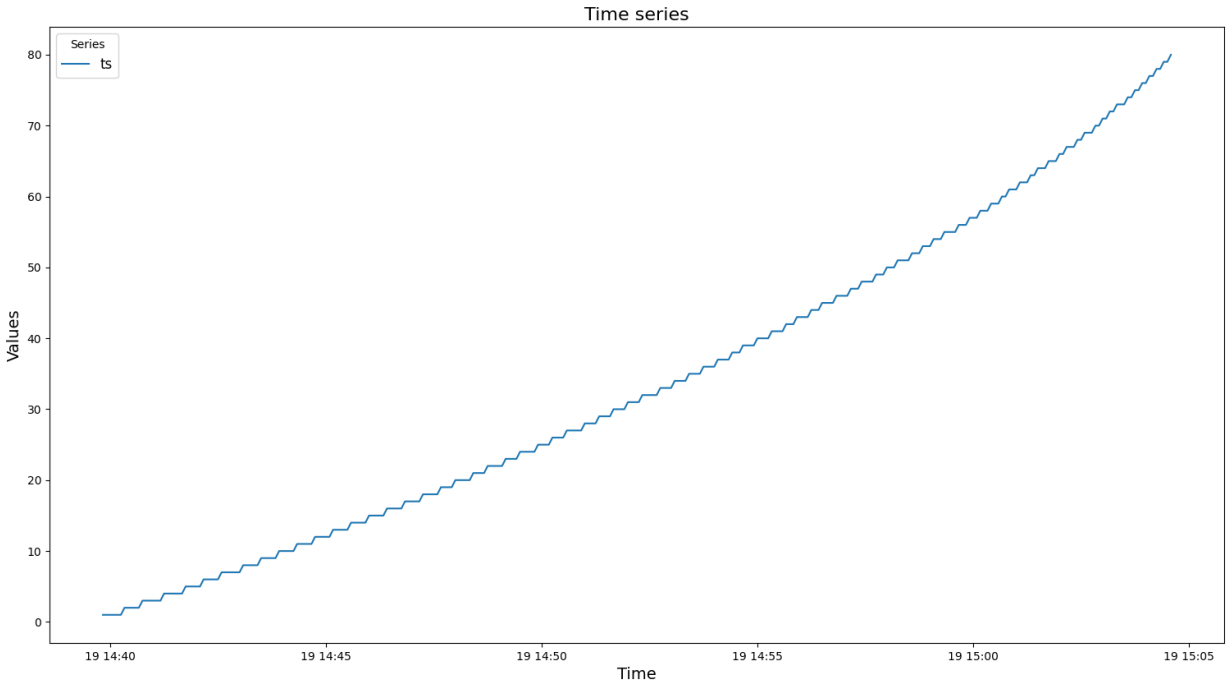
- Facciamo il fit del modello sull'intero data set.
Effettuiamo una predizione di 72 campioni, ovvero relativa ai prossimi 6 minuti.

```
model = ARIMA(trend, order=(15,3,0))
results = model.fit()
fcast = results.predict(len(trend), len(trend)+72, typ='levels')
```

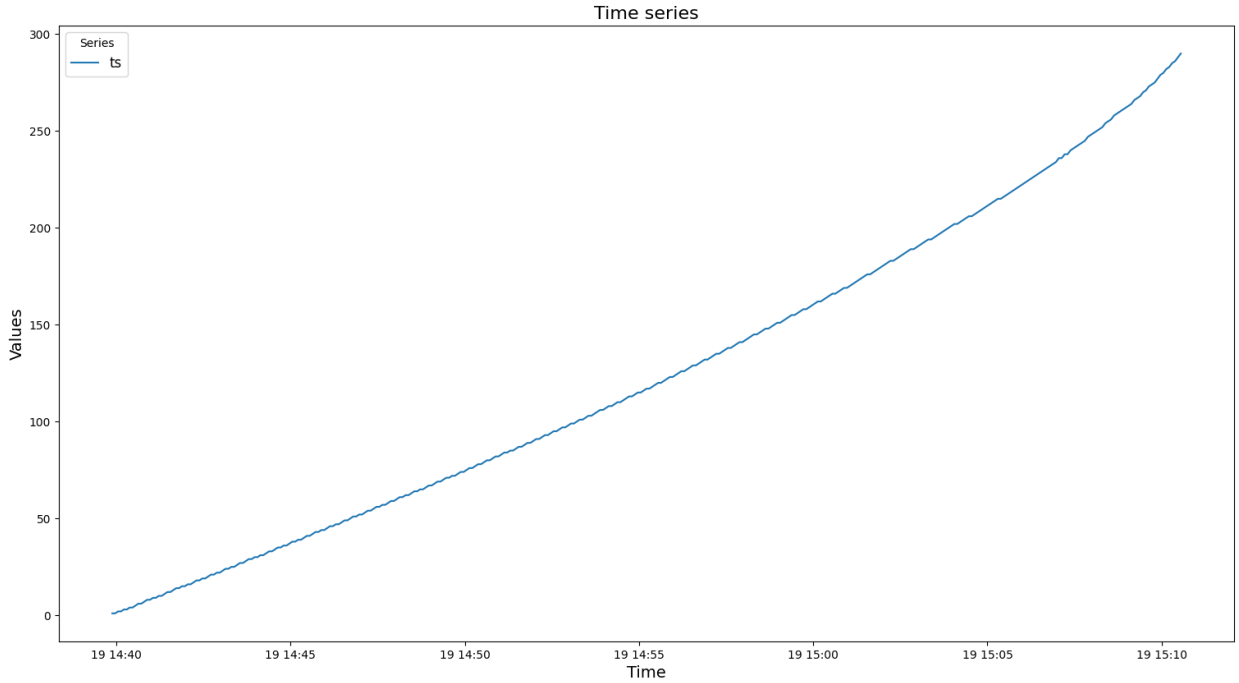


- policyCounter [ts_analysis-policy.py], purchaseCounter [ts_analysis-purchase.py]

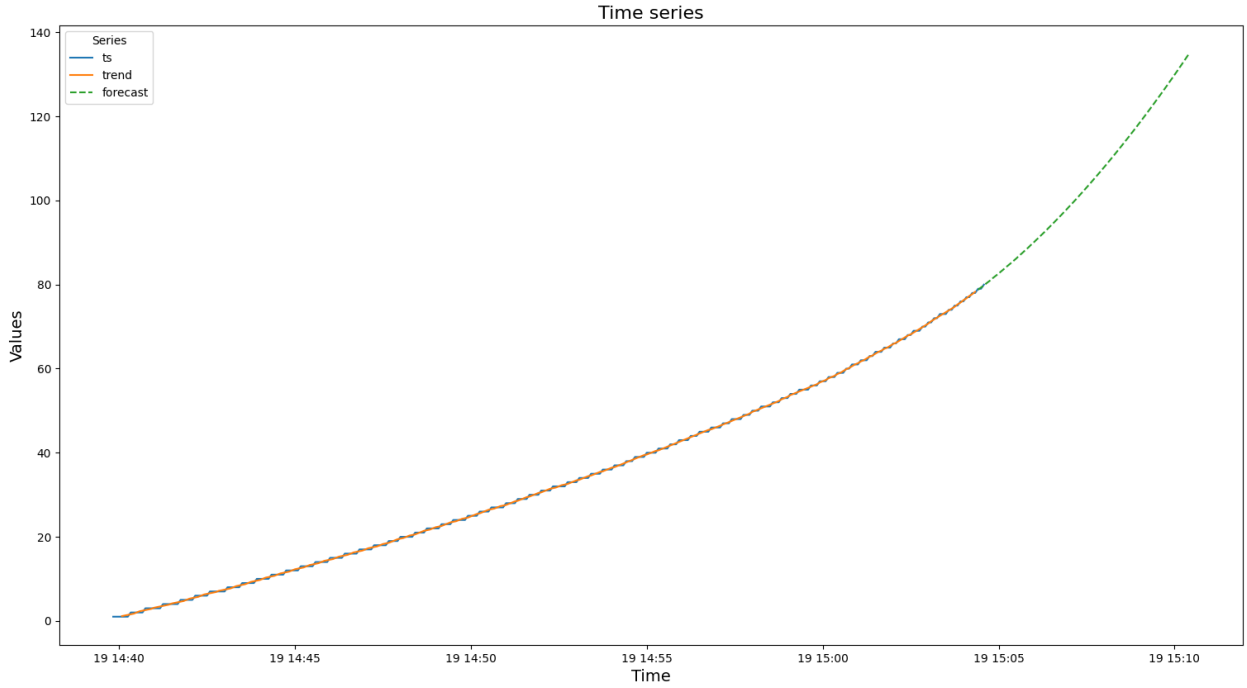
L'analisi delle time series relative alle metriche policyCounter e purchaseCounter risulta essere molto simile a quella effettuata per la metrica userCounter. Ciò è dovuto principalmente al fatto che, seppur il data set sia diverso, i dati presentano in tutti e tre i casi un andamento di tipo esponenziale.
L'analisi completa relativa a queste due metriche può essere visionata nei due file sopracitati.



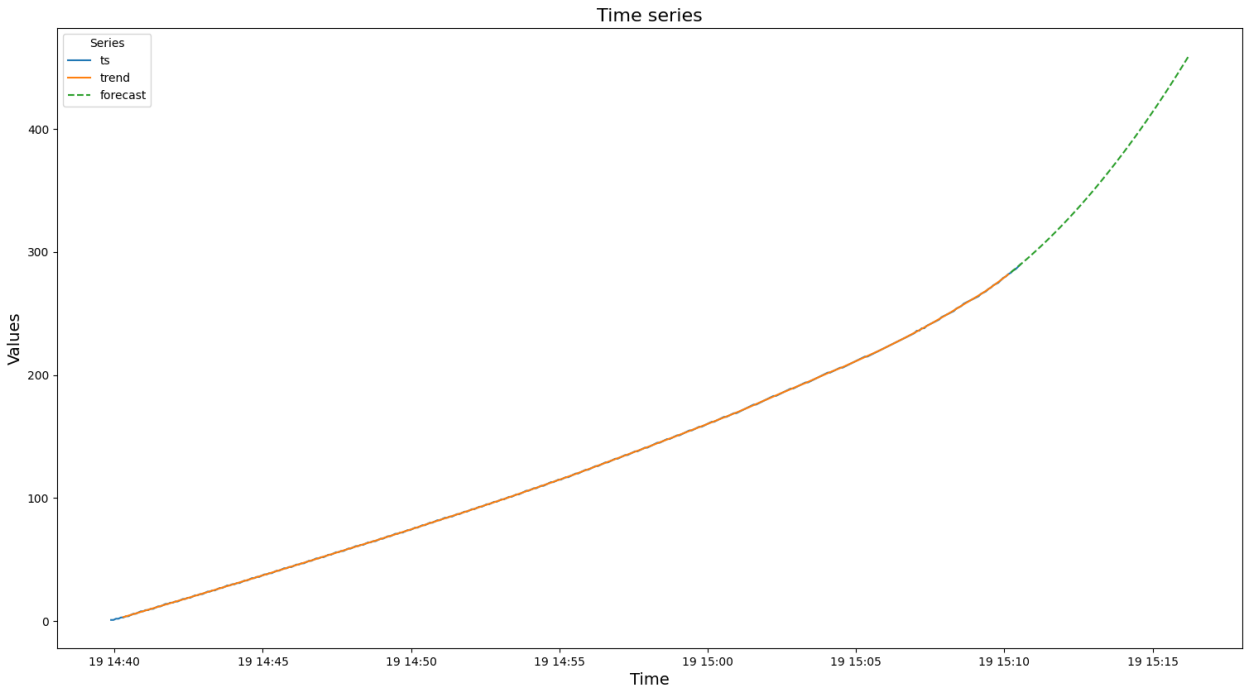
[time series - policy]



[time series - purchase]



[policy - forecasting]



[purchase - forecasting]

Esempio di applicazione reale:
in seguito a queste analisi, se sulla base di una predizione il numero di acquisti che si tenta di effettuare supererà una soglia fissata, potrà essere inviato un alert e di conseguenza potranno essere prese delle contromisure.