Università degli Studi di Milano Bicocca
**Scuola di Scienze**
**Dipartimento di Informatica, Sistemistica e Comunicazione**
**Corso di laurea in Informatica**

# Strumento di refactoring automatico per architetture a microservizi

**Relatore:** *Prof.ssa Francesca Arcelli Fontana*

**Correlatore:** *Dott. Paolo Bacchiega*

**Relazione della prova finale di:**
*Francesco Ambrogio Marinoni*
*Matricola 869276*

**Anno Accademico 2022-2023**

# Indice

# 1 Introduzione

Le architetture a microservizi hanno acquisito grande importanza negli anni e mantengono tutt'ora un forte interesse. Questa tipologia di architetture favorisce la creazione di sistemi informatici più adattabili e robusti[1]. Una qualità che permea le architetture a microservizi è la ridotta dipendenza tra i suoi componenti, caratteristica che consente la modifica e la sostituzione di un componente con un altro senza alterare il funzionamento del sistema. In aggiunta, la maggiore scalabilità del sistema è garantita dalla possibilità di apportare modifiche ad un singolo servizio senza influire sugli altri, riducendone il rischio di regressioni e semplificando la gestione bug e correzioni[2]. L'interesse suscitato dalle architetture a microservizi ha generato anche una spinta verso la conversione di sistemi monolitici in sistemi a microservizi[3]. Gli strumenti che permettono tale conversione sono soluzioni efficienti per la migrazione e l'adattamento delle applicazioni esistenti. In questo contesto, i sistemi di conversione giocano un ruolo fondamentale nel favorire l'adozione diffusa delle architetture a microservizi nell'ambito dello sviluppo software (alcuni esempi possono essere GKE[4], Istio[5] e Apigee[6]). La conversione da architetture monolitiche a microservizi non è un processo perfetto, a causa delle differenze nei paradigmi di progettazione e gestione nelle interazioni. Questa transizione può comportare sfide legate alla decomposizione delle funzionalità esistenti in servizi più piccoli, alla gestione delle dipendenze tra i servizi e alla necessità di implementare nuove pratiche di sviluppo e di distribuzione. Risulta fondamentale, pianificare attentamente la migrazione o la creazione di sistemi a microservizi, poiché senza uno sviluppo controllato, potrebbero essere generati microservice smells (MS)[7]. I MS sono segnali di potenziali problemi nella progettazione o nell'implementazione dei microservizi, come ad esempio una dipendenza stretta, una comunicazione poco efficiente o una distribuzione disomogenea del carico di lavoro. Rilevare e affrontare i MS, risulta cruciale per garantire qualità come la scalabilità e la robustezza delle architetture a microservizi.

Nel contesto delle crescenti esigenze di migrazione e adattamento dei sistemi monolitici alle architetture a microservizi, questa tesi ha condotto alla creazione di AMARI: *Automatic Microservices Architecture Refactoring Instrument*. Partendo da un file .graphml, contenente la struttura di una architettura a microservizi in forma di grafo, AMARI ne consente la visualizzazione e la manipolazione, con lo scopo di rimuovere eventuali MS smells presenti. In seguito, AMARI genera un file .json che contiene un insieme di tracce in forma atomica che rappresentano le comunicazioni del sistema a microservizi secondo lo schema chiave-valore di Zipkin V2. AMARI si prefigge di apportare un significativo contributo alla pianificazione e all'attuazione di architetture a microservizi, ottenendo sistemi che siano scalabili e resilienti.

La tesi è organizzata nei seguenti capitoli: un primo capitolo introduttivo al background teorico e allo stato dell'arte, ai fini della comprensione dei concetti base. Nel capitolo 3, vengono esaminati i .graphml e i .json, due strutture dati fondamentali per immagazzinare le informazioni relative ai sistemi a microservizi, fornendo una comprensione completa di entrambe sia dal punto di vista teorico che pratico. Nel capitolo 4 viene fornita una descrizione dettagliata di AMARI e del suo funzionamento. Nel capitolo 5 sono esaminate le applicazioni pratiche di AMARI attraverso l'analisi di progetti specifici su cui è stato applicato questo strumento. Infine, nel capitolo 6, si espongono le conclusioni e le possibili future estensioni dello strumento presentato.

# 2 Background

## 2.1 Architettura a Microservizi

I microservizi rappresentano un approccio all'architettura del software in cui l'applicazione è suddivisa in servizi indipendenti di dimensioni ridotte, che comunicano tra loro attraverso chiare API. [1]. Le architetture a microservizi sono costituite da sistemi indipendenti chiamati microservizi. Questi, in linea teorica, operano in modo autonomo come Black Box, senza dipendenze strette dai restanti componenti del sistema, al fine di garantire l'intercambiabilità e la scalabilità. Questi sistemi Black Box dipendono da input esterni per le loro operazioni e producono output in base a essi. Questo facilita l'individuazione di problemi nei nodi problematici e la rapida modifica dei microservizi obsoleti o non completamente funzionanti. Questa tesi si concentra sul confronto tra **strutture a microservizi** e "**monoliti**".

## 2.2 Architetture a Microservizi VS Architetture Monolitiche

L'approccio del "monolite" era inizialmente preferito poiché non esistevano molte altre alternative per la costruzione di sistemi, e si basava sulla creazione di un singolo componente che gestiva un'intera serie di azioni dall'inizio alla fine. Nell'eventualità di una richiesta eccessiva o di qualsiasi problema relativo a una delle richieste o a uno dei servizi interconnessi nel "**monolite**", si presenterebbero molteplici complicazioni nel poter aggiornare e modificare quella parte specifica del sistema che presenta errori, con la conseguente perdita della possibilità di avere interscambiabilità a livello di codice e sistema. Tale complessità limita la sperimentazione e rende più difficile implementare nuove idee. Le architetture monolitiche rappresentano un ulteriore rischio per la disponibilità dell'applicazione, poiché la presenza di numerosi processi dipendenti e strettamente collegati aumenta l'impatto di un errore in un singolo processo[1]. Con una **struttura a microservizi** invece risulta che i servizi stessi siano divisi tra di loro e si interconnettano tramite un'interfaccia, o API Gateway. Quanto detto precedentemente può essere confermato anche dalla seguente affermazione, sempre tratta dal sito di **AWS** [1]: "Poiché eseguito in modo indipendente, ciascun servizio può essere aggiornato, distribuito e ridimensionato per rispondere alle richieste di funzioni specifiche di un'applicazione."

L'azione di conversione però non si ferma solo ai monoliti ma ha avuto ripercussioni anche su altre architetture, come ad esempio:

- **Architettura a strati**: In questa architettura, l'applicazione è suddivisa in diversi strati logici, ognuno dei quali svolge una funzione specifica.
- **Architettura a pipeline**: In questa struttura, i dati vengono elaborati in una sequenza di passaggi o fasi, chiamati "pipeline".
  Ogni fase svolge una specifica operazione sui dati e il risultato viene passato alla fase successiva.
- **Architettura a eventi**: Questa architettura si basa sulla comunicazione tra componenti attraverso eventi.
  I componenti generano ed elaborano eventi, consentendo la creazione di sistemi altamente scalabili e reattivi.
- **Architettura a servizi**: Anche se può essere collegata ai microservizi, è una struttura in cui le diverse funzioni o servizi dell'applicazione sono organizzati come componenti indipendenti e riusabili.

- **Architettura a federazione**: Questa struttura coinvolge la combinazione di diversi sistemi o applicazioni indipendenti, ma correlati, in un'unica entità. Ciascun sistema mantiene la propria autonomia, ma può comunicare con gli altri sistemi per scambiare informazioni e dati.
- **Architettura a batch**: Questa architettura riguarda l'elaborazione di grandi quantità di dati in batch, ovvero insiemi di dati raccolti in un intervallo di tempo specifico.

## 2.3 Analisi Dinamica

Normalmente, ci si riferisce all'**analisi dinamica** come al processo di valutazione di un sistema software o di un componente del software stesso, basato sull'osservazione del suo comportamento in esecuzione. Per condurre l'analisi, è richiesto un Oracolo [8], che può essere un componente automatizzato o una persona, che ha conoscenza completa sull'esecuzione di ciascun caso di test al fine di valutare il comportamento del codice. L'**analisi dinamica** procede, quindi, con un preventivo **testing** che servirà a estrarre dei valori che, nella fase successiva cioè la **valutazione**, può **convalidare** o **verificare** il codice.
La convalida è un'attività essenziale che mira a garantire che un'applicazione software soddisfi in modo efficace e accurato gli obiettivi per i quali è stata creata, così come i requisiti e gli scopi previsti. Questo processo è fondamentale per garantire che l'applicazione sia funzionale, affidabile e in grado di risolvere i problemi specifici per cui è stata progettata. La convalida coinvolge un'analisi approfondita dell'applicazione in questione, durante la quale vengono esaminati vari aspetti. Prima di tutto, si verifica se l'applicazione esegue correttamente le funzioni per le quali è stata progettata. In altre parole, si tratta di accertarsi che tutte le caratteristiche e le funzionalità previste funzionino come previsto e siano in grado di risolvere i problemi specifici che l'applicazione dovrebbe affrontare.
La **verifica** si concentra sull'esame dei requisiti formali e specifici dell'applicazione, distinguendosi dall'approccio precedente che aveva a che fare con dati di natura informale.
Quindi in conclusione la verifica si concentra sulla conformità ai requisiti formali del software, mentre la convalida valuta l'efficacia e l'adeguatezza dell'applicazione rispetto alle esigenze degli utenti; la prima verifica se l'applicazione è stata costruita correttamente, mentre la seconda verifica se è stata costruita nel modo giusto per risolvere i problemi degli utenti. Ci sono due categorie di risultati che possono essere ottenute: anomalie che necessitano di correzioni ed errori legati all'affidabilità del prodotto.
Per la **valutazione dinamica** esistono due approcci: il **testing funzionale** o quello **strutturale**.
l **testing funzionale**, noto anche come black box testing, consiste nell'osservare il comportamento esterno del software. Come suggerisce il nome black box, questo tipo di testing non può "vedere" cosa accade internamente, ma valuta solo l'output a partire da un input specifico. Quest'ultimo presenta due vantaggi principali:

- senza conoscere il codice di ciò che si sta analizzando si possono comunque valutare i risultati;
- avendo degli input studiati e degli output attesi, se sia gli output attesi che quelli effettivi coincidono allora il software non presenta errori.

Il **testing strutturale**, noto anche come white box testing, richiede una conoscenza dettagliata del codice sorgente e della struttura del software, a differenza del black box testing, che si basa esclusivamente sul comportamento esterno senza richiedere tale conoscenza. Con il testing strutturale, è possibile eseguire test in modo automatizzato che si fondano su routine specifiche del software. Queste routine possono essere implementate utilizzando il linguaggio del software stesso o altri linguaggi compatibili. Quest'ultimo metodo che agisce sul codice è finalizzato alla copertura di istruzioni, procedure

e parti di codice stesso che compongono strutture.
Il **white box testing** presenta vantaggi significativi, come una risoluzione più agevole dei problemi (debugging) grazie a un testing dettagliato. Quest'ultimo tuttavia, richiede obbligatoriamente un programmatore esperto per essere adeguatamente sviluppato.

## 2.4 Teoria dei grafi

I grafi rivestono un ruolo essenziale in questa tesi. Si tratta di un concetto fondamentale per la gestione dei sistemi a microservizi. Gli alberi, che per definizione sono grafi non orientati, connessi e aciclici, sono particolarmente rilevanti in questo contesto. Questi alberi sono utilizzati per visualizzare chiaramente le diverse dipendenze presenti nei sistemi a microservizi, comprese quelle legate alla comunicazione, alla configurazione, all'accesso ai dati e all'orchestrazione.

### 2.4.1 Grafi

In informatica, un **grafo** è una struttura dati che rappresenta una collezione di nodi (o vertici) interconnessi da archi. L'utilizzo dei grafi è esteso per la rappresentazione di connessioni tra oggetti o entità in svariati scenari, tra cui reti informatiche, algoritmi di ricerca, analisi dei dati e la rappresentazione di sistemi complessi, tra cui le architetture basate su microservizi[9]. Formalmente, un **grafo** può essere definito come una coppia ordinata **G = (V, E)**, in cui V rappresenta l'insieme di vertici ed E l'insieme degli archi. I grafi possono essere caratterizzati da diversi attributi, tra cui la loro natura orientata o non orientata, la presenza o l'assenza di cicli e la loro connettività. Un grafo non orientato prende questo nome perché gli archi che lo compongono possono essere attraversati in entrambe le direzioni, stabilendo così una relazione di simmetria tra i nodi. Un **grafo orientato** è il reciproco di un non orientato e quindi avrà una relazione unidirezionale. Un grafo ciclico ottiene questa designazione quando include almeno una sequenza di archi che forma un ciclo o un percorso chiuso all'interno della sua struttura. L'antitesi di un grafo ciclico è rappresentata da un grafo aciclico, che prende comunemente il nome di DAG (Directed Acyclic Graph) quando oltre a essere privo di cicli è anche orientato, ossia quando le relazioni tra i nodi seguono una direzione specifica senza formare circuiti chiusi. Infine, un **grafo connesso** è tale se esiste un percorso tra una qualsiasi coppia di nodi. Uno dei punti di forza dei grafi è la loro capacità di essere rappresentati attraverso diverse strutture, che sono particolarmente adatte per la loro alta flessibilità. Un esempio notevole di queste strutture è rappresentato dalle liste di adiacenza.

### 2.4.2 Grafi Diretti

Un grafo diretto costituisce una struttura dati che comprende un insieme di nodi, noti anche come vertici, interconnessi tramite archi diretti e orientati. In questo contesto specifico, è fondamentale sottolineare che ogni arco all'interno del grafo rappresenta una relazione unidirezionale ben definita che si estende dal nodo sorgente al nodo destinazione. La chiara direzione degli archi sottolinea la natura univoca delle relazioni tra i nodi all'interno del grafo. Questa direzione conferisce al **grafo** una chiara asimmetria nelle relazioni tra i nodi. In aggiunta, è essenziale notare che in un grafo diretto, la presenza di cicli è esclusa; ciò significa che non è possibile definire un percorso che conduca nuovamente al nodo di partenza attraverso gli archi diretti, garantendo così una struttura aciclica in questo tipo di grafo. Uno dei casi più noti è rappresentato dai **DAG (Directed Acyclic Graph)** che sono un tipo di **grafo diretto** in cui non esistono cicli, ovvero non è possibile seguire una sequenza di archi che riporti al nodo di partenza. Un **DAG** rappresenta una struttura di dipendenza tra nodi, dove un nodo

può dipendere da uno o più nodi precedenti. Questa particolare caratteristica rende i DAG (Directed Acyclic Graph) strumenti di grande utilità quando si tratta di modellare processi di calcolo parallelo, implementare ordinamenti topologici e rappresentare gerarchie di dipendenze in modo efficiente.

## 2.5 Tracce

Una Traccia rappresenta un percorso sequenziale di nodi all'interno di un albero, che inizia dalla sorgente e si estende fino a raggiungere qualsiasi punto all'interno dell'albero stesso(esempio Figura 1).

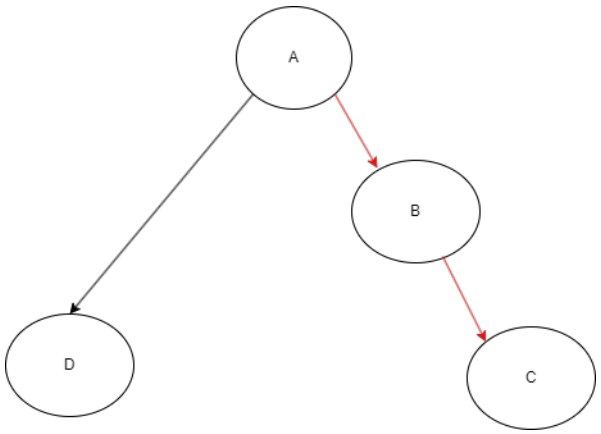


Figura 1: Quella evidenziata in rosso è un esempio di traccia

Riflettendo sulla Figura 1, è possibile affermare che essa rappresenta un albero, in cui ciascun nodo può essere collegato a zero o più nodi figli, con l'unica eccezione del nodo radice che non ha un genitore ed è l'unico caso in cui si verifica questa condizione.
Le **tracce** di un albero possono trovare applicazioni in diversi contesti, come ad esempio, la navigazione o la ricerca all'interno di una struttura ad albero.

## 2.6 Differenza tra tracce e metriche

Le **tracce**, come descritte nella sottosezione precedente, si riferiscono ai percorsi costituiti dai nodi di un albero. Inoltre, rappresentano le sequenze di nodi che si possono seguire per raggiungere un nodo specifico. Le **tracce** sono utili per la navigazione e la ricerca all'interno di una struttura ad albero. Le **metriche**, per contro, sono misurazioni quantitative utilizzate per valutare o quantificare specifiche caratteristiche o prestazioni di un sistema. Le **metriche** forniscono dati numerici che possono essere utilizzati per misurare la qualità, l'efficienza, l'efficacia o altre proprietà di un sistema. Le **tracce** e le **metriche** sono entrambi strumenti utili per monitorare e gestire un sistema a microservizi. Le **tracce** possono essere utilizzate per individuare il percorso di una richiesta attraverso i diversi microservizi che compongono il sistema. Quando una richiesta viene ricevuta da un microservizio, possono essere registrate informazioni come l'identificatore univoco della richiesta, il tempo di arrivo, il microservizio corrente e il tempo trascorso nel microservizio stesso. Queste informazioni possono

essere utili per la diagnostica, il monitoraggio e l'analisi delle prestazioni complessive del sistema. Ad esempio, le **tracce** possono essere utilizzate per individuare i tempi di risposta lenti o i possibili "collo di bottiglia" nell'elaborazione delle richieste. Le **metriche**, invece, possono essere utilizzate per misurare e valutare le prestazioni, la scalabilità, l'affidabilità e altri aspetti del sistema a microservizi. Quest'ultime possono includere misurazioni come il tempo di risposta medio di un microservizio, il numero di richieste elaborate al secondo, l'utilizzo della CPU o la quantità di memoria utilizzata da ogni microservizio. Le **metriche** possono essere raccolte e monitorate continuamente per identificare i problemi di prestazioni, la capacità dei microservizi, rilevare l'aumento del carico o l'impatto delle modifiche apportate al sistema. Le **metriche** possono anche essere utilizzate per impostare soglie o avvisi che segnalano eventuali anomalie o superamenti dei limiti desiderati. In entrambi i casi, **tracce** e **metriche** forniscono informazioni preziose per il monitoraggio e la gestione di un sistema a microservizi. Possono essere utilizzate per ottimizzare le prestazioni, identificare problemi, migliorare la scalabilità e fornire un'analisi dettagliata sul comportamento del sistema nel suo complesso.

## 2.7 Gli Strumenti Coinvolti

In questa sezione, verranno esaminati diversi strumenti che sono stati coinvolti in modo pratico, per esempio testing risultati, e non nella costruzione di AMARI. L'obiettivo è fornire una comprensione più approfondita del modo in cui il tool proposto in questa tesi applicherà le funzioni che sono state implementate. Questa panoramica dei tool è fondamentale per una migliore comprensione del contesto e delle applicazioni pratiche delle soluzioni proposte. Uno dei tool è **Aroma** [7]; strumento finalizzato per la ricostruzione automatica di architetture e per trovare gli **smell** correlati alle stesse, basato sull'analisi dinamica di tracce di esecuzione di microservizi. Tramite il **reverse engineering** il tool ha la possibilità di analizzare le applicazioni a **microservizi** e costruire il grafo di dipendenza delle architetture inerente ai sistemi a **microservizi** stessi e alle loro dipendenze. Un altro dei tool è **Zipkin**[10]; strumento **open-source** progettato per fornire un sistema di tracciamento delle richieste distribuite in ambienti di **microservizi**. Questo strumento è particolarmente utile per individuare e risolvere problemi di prestazioni e latenza all'interno di un'**architettura a microservizi** complessa. **Zipkin** opera con il principio del tracciamento distribuito. Ogni richiesta inviata tra i **microservizi** è dotata di un **ID** di tracciamento univoco, consentendo a **Zipkin** di seguire il percorso completo della richiesta attraverso la rete di servizi. Questi **ID** di tracciamento sono collegati a una serie di **"span"** che rappresentano il tempo trascorso all'interno di ogni servizio durante l'esecuzione della richiesta. La restituzione delle "**span**" è in un file di formato **JSON**. Un altro tool è **Jaeger**[11]; strumento **open-source** di tracciamento distribuito, progettato per monitorare e risolvere problemi di prestazioni all'interno di ambienti di **microservizi complessi**. Il principale obiettivo di **Jaeger** è fornire un sistema di tracciamento distribuito che permetta di visualizzare e analizzare il flusso delle richieste attraverso l'intera **architettura dei microservizi**. Ogni richiesta è dotata di un identificatore unico, chiamato "**trace ID**", che viene propagato attraverso i vari servizi coinvolti nella gestione della richiesta. **Jaeger** è in grado di raccogliere informazioni dettagliate su ogni "**span**" all'interno di un **trace**, inclusi i tempi di esecuzione, le dipendenze tra i servizi e i possibili errori o latenze. Un altro tool è **yEd**[12]; abbreviazione di "**yEd Graph Editor**", è uno strumento software di grafica vettoriale sviluppato da **yWorks GmbH**. Si tratta di un'applicazione versatile e potente utilizzata principalmente per la creazione, la modifica e l'analisi di diagrammi e **grafi**. Questo strumento è ampiamente utilizzato in diversi campi, come la gestione dei progetti, l'analisi dei dati, la modellazione di processi e la visualizzazione delle reti.

# 3 Gestione dei Grafi con GraphML e JSON

Questo capitolo si concentra sulla gestione dei dati di grafi utilizzando i formati GraphML e JSON.Nei capitoli 3.1 e 3.2, vengono esplorate le caratteristiche del formato GraphML, ampiamente utilizzato per rappresentare grafi diretti e non diretti, permettendo la memorizzazione di informazioni complesse come attributi sui nodi e sugli archi. Verrà mostrato come è possibile modificare, analizzare e visualizzare grafi utilizzando GraphML. Successivamente, nei capitoli 3.3 e 3.4, verrà esaminata la sinergia tra GraphML e JSON, evidenziando come i file JSON possano essere utilizzati per conservare e condividere dati rappresentati in formato GraphML. Sarà illustrato come la conversione tra questi due formati possa semplificare la gestione dei dati di grafi, permettendo un'interoperabilità più fluida e l'integrazione in applicazioni che supportano JSON. Nella sottosezione 3.4.1, verrà analizzato il processo di trasposizione dei dati nel formato JSON, seguendo lo stile di Zipkin. Questa operazione è essenziale per migliorare la portabilità e l'accessibilità dei dati di tracciamento delle applicazioni, specialmente considerando l'importante legame tra AMARI e Aroma, che richiede la disponibilità dei dati nel formato specifico di Zipkin JSON per la corretta integrazione e comunicazione tra le due piattaforme. Nell'ultima sottosezione del capitolo, verrà esplorato in dettaglio come generare le tracce di dati per il file JSON stile Zipkin stesso.

## 3.1 GraphML

Il **GraphML (Graph Markup Language)**[13] è un formato di file basato su XML (eXtensible Markup Language) impiegato per la rappresentazione di dati di tipo grafo e delle relative strutture. I grafi rappresentano strutture dati costituite da nodi (o vertici) e archi (o spigoli) che collegano tra loro i nodi. GraphML fornisce una sintassi normalizzata per la descrizione dei grafi in un formato facilmente leggibile sia per gli esseri umani che per le macchine. La sua struttura **XML** consente di rappresentare in modo dichiarativo informazioni sulle proprietà dei nodi, degli archi e del **grafo** nel suo complesso. Queste informazioni possono includere etichette, attributi, pesi, colori, tipi di dato e altro ancora. Il formato **GraphML** è molto flessibile e adatto per rappresentare **grafi** di varie dimensioni e complessità. Grazie alla sua natura estensibile, GraphML consente l'inclusione di elementi personalizzati o specifici per un determinato dominio, rendendolo una scelta ideale come formato per rappresentare i sistemi a microservizi. I **GraphML** si compongono di diversi attributi, nel caso di AMARI, gli attributi richiesti sono(Figura 2):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
  xmlns:java="http://www.yworks.com/xml/yfiles-common/1.0/java"
  xmlns:sys="http://www.yworks.com/xml/yfiles-common/markup/primitives/2.0"
  xmlns:x="http://www.yworks.com/xml/yfiles-common/markup/2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:y="http://www.yworks.com/xml/graphml"
  xmlns:yed="http://www.yworks.com/xml/yed/3"
  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
    http://www.yworks.com/xml/schema/graphml/1.1/ygraphml.xsd">
<!--Created by yEd 3.22-->
<key for="port" id="d0" yfiles.type="portgraphics"/>
<key for="port" id="d1" yfiles.type="portgeometry"/>
<key for="port" id="d2" yfiles.type="portuserdata"/>
<key attr.name="labelV" attr.type="string" for="node" id="d3"/>
<key attr.name="name" attr.type="string" for="node" id="d4"/>
<key attr.name="url" attr.type="string" for="node" id="d5"/>
<key attr.name="description" attr.type="string" for="node" id="d6"/>
<key for="node" id="d7" yfiles.type="nodegraphics"/>
<key for="graphml" id="d8" yfiles.type="resources"/>
<key attr.name="labelE" attr.type="string" for="edge" id="d9"/>
<key attr.name="weight" attr.type="int" for="edge" id="d10"/>
<key attr.name="isDB" attr.type="boolean" for="edge" id="d11"/>
<key attr.name="url" attr.type="string" for="edge" id="d12"/>
<key attr.name="description" attr.type="string" for="edge" id="d13"/>
<key for="edge" id="d14" yfiles.type="edgegraphics"/>
<graph edgedefault="directed" id="G">
```

Figura 2: Esempio GraphML

Come evidenziato nella seconda riga di codice(Figura 2), l'estratto completo degli attributi del nostro grafo è ottenuto tramite l'utilizzo del tool denominato yEd, il quale è dettagliato nella sezione 2.6 del documento.

## 3.2 Gestione dei GraphML

L'implementazione di AMARI è stata realizzata utilizzando il linguaggio di programmazione Python.
In Python possiamo trovare molteplici librerie per gestire i file **GraphML**:

1. **xml.etree.ElementTree** [1]
2. **xml.dom** [2]
3. **xml.dom.minidom** [3]
4. **xml.dom.pulldom**[4]
5. **xml.sax**[5]
6. **xml.parsers.expat**[6]
7. **networkx**[7]

Dopo aver eseguito test con tutte le suddette librerie Python, si è optato per networkx.
Questa scelta è stata motivata principalmente dal fatto che networkx non richiedeva l'uso di una combinazione di librerie diverse ed offriva comandi chiari e facilmente comprensibili, anche per coloro che non erano familiari con la libreria.
I comandi che sono stati utilizzati all'interno di AMARI, estratti dalla libreria **networkx**[14], sono:

1. **read_graphml** : la funzione che permette di leggere dal file e salvare in una variabile i dati del grafo, tra cui con gli attributi nodes, per i nodi, ed edges, per gli archi.
2. **find_cycle** : per trovare se il grafo presenta cicli.
3. **is_directed_acyclic_graph** : per controllare se il grafo sia diretto e aciclico.
4. **draw** : per rappresentare il grafo in collaborazione con la libreria **matplotlib.pyplot**.
5. **has_path** : per identificare se ci sia almeno un path tra 2 nodi specifici.
6. **all_simple_paths** : per estrapolare tutti i path dalla radice a qualsiasi nodo (con anche ripetizione dei percorsi).

Dopo aver inizializzato nella variabile **g** i dati letti dal **grafo** è stato possibile utilizzare non solo i loro attributi nodes ed edges ma anche dei metodi, sempre presenti nella libreria **networkx**, che si possono adoperare per aggiungere, modificare o eliminare nodi e archi nel **GraphML** e, questi comandi sono:

1. **add_node** : aggiunge un nodo al grafo, se non è già presente.

[1] **xml.etree.ElementTree**
[2] **xml.dom**
[3] **xml.dom.minidom**
[4] **xml.dom.pulldom**
[5] **xml.sax**)
[6] **xml.parsers.expat**
[7] **networkx**

2. **add_edge** : aggiunge un arco tra due nodi specifici, se i nodi non sono esistenti l'arco non viene creato.

3. **remove_node** : rimuove un nodo dal grafo, se esistente nel grafo.

4. **remove_edge** : rimuove un arco tra due nodi specifici, se l'arco è esistente.

Per rendere possibile una modifica efficace del grafo, è stato necessario mantenere invariati i metodi di aggiunta e rimozione dei nodi. Tuttavia, per quanto riguarda la creazione degli archi, abbiamo dovuto apportare delle modifiche al sistema, implementando un controllo per verificare l'esistenza dei nodi. Nel caso in cui i nodi non fossero presenti, il tool offriva l'opzione di crearli in modo da garantire un processo più completo. Per le modifiche di archi nel **grafo** si è dovuto applicare una combinazione di metodi e ciò lo si può catalogare in questa serie di passaggi:

1. controllare se l'arco esiste

2. l'arco esiste quindi il tool chiederà quale nodo si vuole modificare e in che senso l'arco debba essere orientato; se il nodo non esiste permette di crearlo

3. se si modifica il secondo nodo chiede se esso sia un nodo database

## 3.3 JSON

Un file **JSON (JavaScript Object Notation)** è un formato di dati leggero e intercambiabile utilizzato per rappresentare informazioni strutturate in maniera organizzata. "I dati sono sovrani. Ma sapere come lavorare con una varietà di dati è diventato ancora più importante. Programmatori, sviluppatori e professionisti IT devono trasferire strutture di dati popolate da qualsiasi linguaggio a formati riconoscibili da altri linguaggi e altre piattaforme. JavaScript Object Notation (JSON) è il formato di scambio dati che rende possibile tutto ciò.", come confermato dal sito di Oracle[15]. JSON ha guadagnato popolarità tra gli sviluppatori grazie alla sua leggibilità e al suo formato leggero, il che si traduce in una minore necessità di codifica e in un'elaborazione più rapida dei dati. JSON è basato sulla sintassi degli oggetti JavaScript, ma può essere utilizzato con successo in molti altri contesti oltre alla trasmissione dei dati. Ad esempio, può essere utilizzato per la configurazione, il salvataggio di dati strutturati e altro ancora. I file **JSON** sono costituiti da coppie chiave-valore, dove le chiavi sono stringhe e i valori possono essere tipi di dati come stringhe, numeri, booleani, array o altri oggetti **JSON** nidificati. I dati sono strutturati in un formato che risulta facilmente leggibile sia per gli esseri umani che per le macchine. Per questa intercambiabilità e possibilità di mantenere un interconnessione ordinata tra elementi si può definire che quindi i file **JSON** siano perfetti per rappresentare le tracce presentate nella sezione 2.3. potresti fornire un esempio specifico di come i dati acquisiti dall'analisi del GraphML vengono tradotti in un formato JSON che rappresenta la struttura del sistema a microservizi.

## 3.4 Trasposizione dei Dati

Prima di esplorare questa sezione, è fondamentale acquisire una comprensione preliminare della struttura di un JSON con codifica Zipkin. Successivamente, dovremo affrontare il tema di come gestire l'ordine e inserire tutti i dati che sono stati catalogati.

### 3.4.1 JSON Zipkin

Nel contesto del formato Zipkin, nei file JSON è necessario che siano presenti alcuni parametri chiave, definiti come obbligatori, e possono essere inclusi anche parametri facoltativi, offrendo quindi una struttura flessibile.
Il JSON sarà costituito da diverse liste, indicate mediante le parentesi quadre, mentre gli elementi saranno contenuti tra parentesi graffe.
Ciascuna di queste liste rappresenterà una traccia, e la lista più esterna rappresenterà l'insieme di tutte le tracce possibili.
Gli elementi si compongono di molteplici attributi e questi sono, come nell'esempio a Figura 3:

1. **traceId** : segnala in che traccia è presente e che percorso sta analizzando
2. **id** : è un nome di categorizzazione che si utilizza per riconoscere l'elemento specifico
3. **parentId** : se ha un nodo genitore verrà inserito in questo campo tramite l'id del genitore stesso
4. **timestamp** : è un dato che cataloga quando venga composto l'elemento
5. **duration** : è il tempo di richiesta attuato sull'elemento del sistema
6. **localEndpoint** : è l'insieme dei dati del nodo che cataloghiamo in locale
7. **remoteEndpoint** : è l'insieme dei nodi collegati al nodo in considerazione ma esterni al sistema, ad esempio database.

Questo schema è stato estratto dal sito Swagger.io [16] e successivamente aggiornato con nuove informazioni ottenute attraverso il processo di testing.

```
  {
   "traceId": "trace1",
   "id": "api-gateway_id",
   "timestamp": 1647943646909605,
   "duration": 377,
   "localEndpoint": {
     "serviceName": "api-gateway"
   }
},
```

Figura 3: Esempio JSON formato Zipkin

### 3.4.2 Come creare le tracce

Per generare le tracce, è possibile iniziare utilizzando il comando "all_simple_paths" dalla libreria networkx, il quale consente di estrarre tutti i percorsi disponibili dalla radice a qualsiasi nodo.
Tuttavia, utilizzando questo metodo, è possibile ottenere dei percorsi duplicati. Ad esempio, se avessimo quattro nodi connessi in modo lineare come A, B, C e D, l'output di "all_simple_paths" potrebbe essere il seguente:

1. A
2. A, B
3. A, B, C
4. A, B, C, D

Pertanto, è possibile concludere che questo metodo genera ripetizioni poiché tutti i percorsi precedenti al quarto sono considerati sotto-percorsi dello stesso quarto percorso. Dopo aver identificato tutti questi percorsi, AMARI esegue una valutazione ricorsiva. Inizia con il primo percorso e lo confronta con tutti gli altri percorsi. Se trova almeno un percorso che contiene il percorso attualmente in analisi, lo rimuove dalla lista. Nell'esempio sopra menzionato, le tracce tra 1 e 3 verranno eliminate, e l'unica che sarà mantenuta sarà la quarta traccia

# 4 Funzionamento di AMARI

Sintetizzando l'analisi sia dei file GraphML che dei file JSON, è possibile ottenere una visione completa delle funzionalità offerte dal tool AMARI.

```
> python .\main.py
insert the folder in which is the graphml:
```

Figura 4: Selezione del GraphML

AMARI richiede la specifica del file da analizzare, il quale deve trovarsi nella stessa cartella(Figura 4). Scegliendo un progetto, il tool recupererà automaticamente il file in formato GraphML associato(Figura 5).

```
> python .\main.py
insert the folder in which is the graphml: Sock Shop
Nodes:  ['0', '2', '5', '8', '10', '12', '14', '16', '18']
Edges:  [('0', '10'), ('2', '5'), ('2', '18'), ('2', '0'), ('2', '8'), ('12', '0'),
          ('12', '2'), ('12', '14'), ('14', '16')]
```

Figura 5: Stampa dei nodi di Sock Shop

Come già menzionato nella sezione dei GraphML(sezione 3.1,3.2), AMARI utilizza le librerie NetworkX e Matplotlib.pyplot per visualizzare il grafo(Figura 6). Inoltre, offre la possibilità di selezionare un tipo specifico di smell da controllare all'interno del grafo, consentendo agli utenti di effettuare un'analisi dettagliata in base alle proprie preferenze(Figura 7).

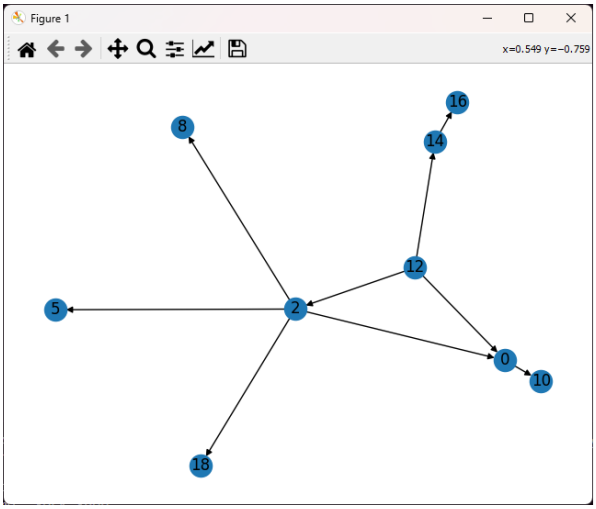


Figura 6: GraphML di Sock Shop

```
> python .\main.py
insert the folder in which is the graphml: Sock Shop
Nodes:  ['0', '2', '5', '8', '10', '12', '14', '16', '18']
Edges:  [('0', '10'), ('2', '5'), ('2', '18'), ('2', '0'), ('2', '8'), ('12', '0'),
         ('12', '2'), ('12', '14'), ('14', '16')]
What do you want to check in the graph? (nga(No Gateway API), sp(Shared Persistence), cd(Cyclic
    Dependency), e(exit))
```

Figura 7: Richiesta da parte di AMARI inerente a quale smell si voglia analizzare

Nel caso in cui uno dei smell sia rilevato, il tool offre l'opportunità di apportare modifiche al grafo attraverso una serie di domande specifiche, presenti nelle figure 7/8/9/10 e 11, identificando i punti critici(Figura 8). Dopo ogni modifica, AMARI presenterà nuovamente il grafo, consentendo ulteriori opportunità per effettuare aggiustamenti o revisioni, se necessario(Figura 9).

```
What do you want to check in the graph? (nga(No Gateway API), sp(Shared Persistence), cd(Cyclic
    Dependency), e(exit)) nga
No API Gateway found
root node: frontend
root id: 12
Problematic nodes: None
Do you wnt to add one or modify the graph? (y/n)
```

Figura 8: Esempio di scelta di controllo presenza del No API Gateway

```
Do you want to add one or modify the graph? (y/n) y
Nodes:  ['0', '2', '5', '8', '10', '12', '14', '16', '18']
Edges:  [('0', '10'), ('2', '5'), ('2', '18'), ('2', '0'), ('2', '8'), ('12', '0'),
         ('12', '2'), ('12', '14'), ('14', '16')]
Do you want to add/delete/modify a node/edge? (y/n) (n to exit)
```

Figura 9: Risposta y per attuare la modifica del grafo

Le modifiche che è possibile fare come visualizzabile nella Figura 10 e già trattati in precedenza nella sezione 3.2 sono:

- an = add_node o aggiungi un nodo
- ae = add_edge o aggiungi un arco(compreso dei due nodi se non presenti in precedenza)
- an = delete_node o elimina un nodo(se presente)
- de = delete_edge o elimina un arco(nodi non compresi)
- mn = modify_node o modifica un nodo
- me = modify_edge o modifica un arco(se non presente uno dei due nodi o entrambi da la possibilità di crearli)

Una volta completate tutte le modifiche, AMARI richiederà all'utente di salvare il file JSON risultante(Figura 11). Si darà la possibilità di scegliere un nome personalizzato per il file, sebbene sia consigliabile salvarlo come "spans" per agevolarne il successivo inserimento in Aroma e l'analisi correlata.

```
Do you want to add/delete/modify a node/edge? (y/n) (n to exit) y
Do you want to add a node or an edge or delete a node or an edge or modify a node or an edge?(an
    /ae/dn/de/mn/me)
```

Figura 10: Stampa scelte possibili di modifica date da AMARI

```
Do you want to add a node or an edge or delete a node or an edge or modify a node or an edge?(an
    /ae/dn/de/mn/me) an
Enter the ID of the node: 300
Enter the name of the node: tester
Enter the name of the node: tester-label
Nodes:  ['0', '2', '5', '8', '10', '12', '14', '16', '18', '300']
Edges:  [('0', '10'), ('2', '5'), ('2', '18'), ('2', '0'), ('2', '8'), ('12', '0'),
         ('12', '2'), ('12', '14'), ('14', '16')]
Do you want to add/delete/modify a node/edge? (y/n) (n to exit) y
Do you want to add a node or an edge or delete a node or an edge or modify a node or an edge?(an
    /ae/dn/de/mn/me) dn
Enter the ID of the node: 300
Nodes:  ['0', '2', '5', '8', '10', '12', '14', '16', '18']
Edges:  [('0', '10'), ('2', '5'), ('2', '18'), ('2', '0'), ('2', '8'), ('12', '0'),
         ('12', '2'), ('12', '14'), ('14', '16')]
Do you want to add/delete/modify a node/edge? (y/n) (n to exit) n
Do you want to modify it again? (y/n)n
Insert the name of the file: spans
```

Figura 11: Stampa scelte possibili di modifica date da AMARI

Il file appena creato sarà automaticamente inserito all'interno della stessa cartella del tool(Figura 12).

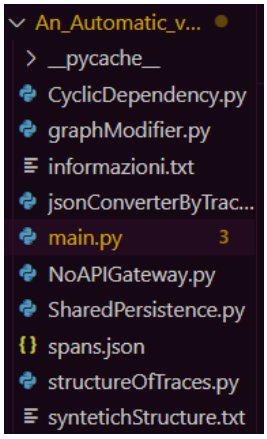


Figura 12: Salvataggio della span.json

## 4.1 Smell riconosciuti dal tool

Basandosi sulla struttura di Aroma[7], AMARI si concentrerà sull'analisi degli stessi smell precedentemente catalogati. Prima di procedere con una disamina più approfondita, definiamo concisamente cosa siano gli smell: essi rappresentano indicatori di potenziali debolezze o pratiche non ottimali nel design di un sistema software, situazioni che potrebbero generare problemi di manutenzione, limitata scalabilità o altre sfide a lungo termine. Nel **tool** vengono analizzati 3 **smells**:

- **Dipendenza Ciclica** : in architetture a microservizi si riferisce a una situazione in cui i servizi dipendono l'uno dall'altro in modo circolare, creando una rete di dipendenze cicliche(Figura 13). Una dipendenza ciclica in un'applicazione di microservizi può danneggiare la capacità dei servizi di scalare o distribuire in modo indipendente, nonché violare il principio di dipendenza aciclica[7]

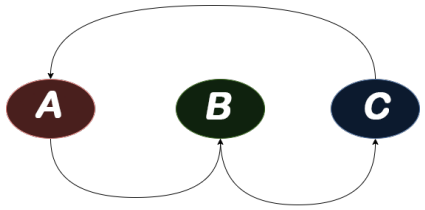


Figura 13: Dipendenza Ciclica

- **No API Gateway** : in architetture a microservizi si riferisce a una situazione in cui i microservizi all'interno del sistema interagiscono direttamente tra loro senza l'uso di un **API Gateway**, che dovrebbe essere la radice del grafo.

  La presenza di questo smell può verificarsi in due diversi scenari:

  - l'**API Gateway** è assente (non presente per progettazione)
  - l'**API Gateway** è presente, ma bypassato da alcune richieste client.

  Si sostiene sia possibile avere fino ai 50 moduli distinti senza dover fare affidamento su un **API Gateway**.[7]

  1. più di un nodo radice: A, B(Figura 14 nuvola 1)
  2. è presente un nodo B isolato dalla radice G(Figura 14 nuvola 2)
  3. è l'insieme della prima casistica e della seconda(Figura 14 nuvola 3)

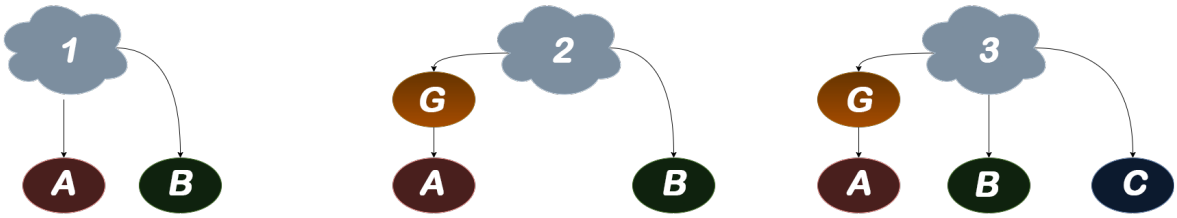


Figura 14: No API Gateway

- **Shared Persistence** : in architetture a microservizi si riferisce a una situazione in cui più microservizi del sistema condividono lo stesso database o la stessa persistenza dei dati.
  Nel peggiore dei casi, diversi servizi accedono alle stesse entità dello stesso database(Figura 15).[7]

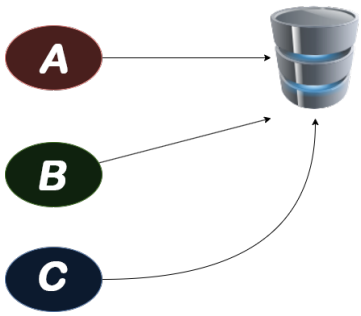


Figura 15: Shared Persistence

Più nodi si collegano al database con differenti serverName label.

## 4.2 Risoluzione degli smell

Per affrontare efficacemente i problemi legati agli smell nei sistemi a microservizi, è essenziale comprendere prima come tali smell influiscano sulla struttura. Inoltre, In questa sottosezione, verranno illustrati i metodi risolutivi per identificare le aree problematiche e le strategie d'intervento. Seguendo i punti della sotto sezione 4.1 identificheremo i problemi di quei determinati **smell** dove siano riscontrati.

1. **Dipendenza ciclica** : il tool segnalerà i punti compresi nel ciclo/i isolandoli in tuple composte dagli ID dei nodi una per ciclo.

2. **No API Gateway** : il tool segnalerà se non è presente una radice interconnessa a tutti i punti e quali sono i punti più interconnessi.

3. **Shared Persistance** : il tool trova tutti i nodi database e li cataloga e segnala se più nodi contattano lo stesso database e per ogni database segnala i due nodi dell'arco problematico.

In questo contesto, AMARI consente di interagire con nodi specifici utilizzando i metodi dettagliati nella sezione 4.0 attraverso una modalità d'uso basata su domande e risposte. Questo processo avviene non mediante l'intervento diretto nel codice, ma piuttosto attraverso una serie graduale di interazioni che si manifestano nella Console o in PowerShell, a seconda del contesto in uso.

# 5 Progetti Originali e Progetti Aggiornati

Nelle prossime sottosezioni, verrà esaminato come l'implementazione di AMARI abbia avuto un notevole impatto su tre progetti chiave: Sock Shop (5.1), Synthetic (5.2) e Online Boutique (5.3). Si prenderanno in considerazione le modifiche apportate a ciascun progetto, evidenziando come AMARI abbia contribuito a migliorare l'efficienza, la scalabilità e le prestazioni complessive di questi sistemi. Sarà affrontata la trasformazione delle architetture, le nuove caratteristiche introdotte e l'impatto generale di AMARI sulla loro operatività. Questo approfondimento consentirà di acquisire una vasta comprensione di come AMARI possa essere applicato con successo in diversi contesti di sviluppo progettuale di sistemi a microservizi.

## 5.1 Progetto SockShop

Uno dei progetti che ha richiesto una ristrutturazione è stato Sock Shop[17]. Se si osserva attentamente la struttura originale del sistema a microservizi (Figura 16),

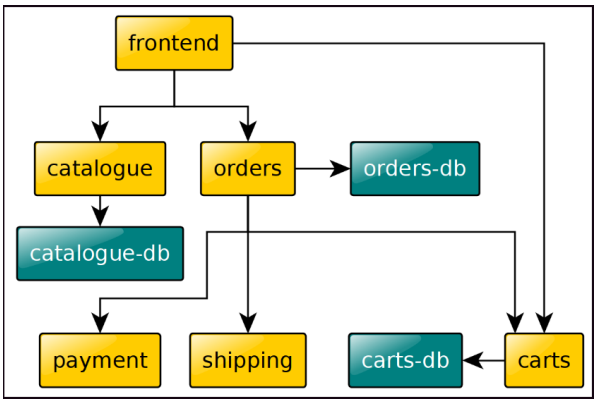


Figura 16: Grafo dello schema di Sock Shop

Tuttavia, nell'analisi del grafo prodotto da Aroma, ci si limita attualmente all'osservazione dello schema (Figura 17).

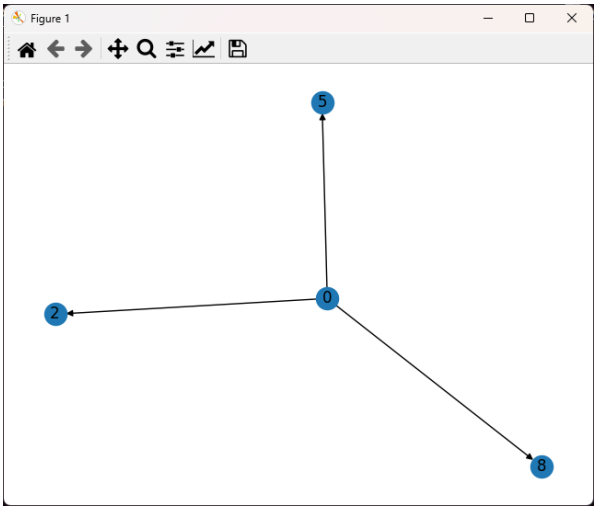


Figura 17: GraphML allo stato iniziale di Sock Shop

I nodi che si possono osservare stampati sono nella Figura 17:

- 0 = carts
- 2 = orders
- 5 = payment
- 8 = shipping

Dopo aver individuato i nodi presenti mediante la funzionalità di modifica del tool, si è proceduto all'inserimento dei nodi mancanti (Figura 18).

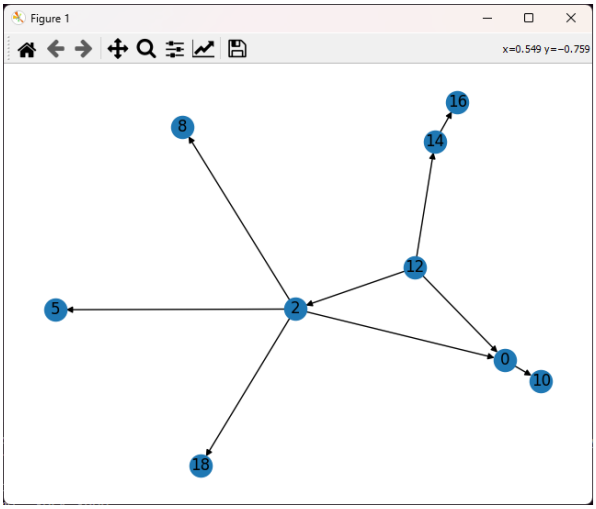


Figura 18: GraphML dopo le modifiche di Sock Shop

I nodi aggiunti sono(Figura 18):

- 10 = carts-db
- 18 = orders-db
- 12 = frontend
- 14 = catalogue
- 16 = catalogue-db

Di conseguenza, mediante queste aggiunte, è stato possibile ricostruire la struttura del sistema a microservizi di Sock Shop e valutare se il sistema presentasse eventuali smells che potessero essere successivamente risolti dallo stesso strumento.
L'aggiunta dei nodi precedenti in questo progetto ha affrontato un problema di mancanza di un API Gateway, il quale è stato risolto grazie all'integrazione di tutti i nodi del sistema con il frontend come radice dell'albero/grafo.

## 5.2 Progetto Synthetic

Un altro progetto che è stato soggetto a ricostruzione è stato Synthetic, in quanto la struttura originale del sistema a microservizi (Figura 19) mostrava la necessità di interventi.

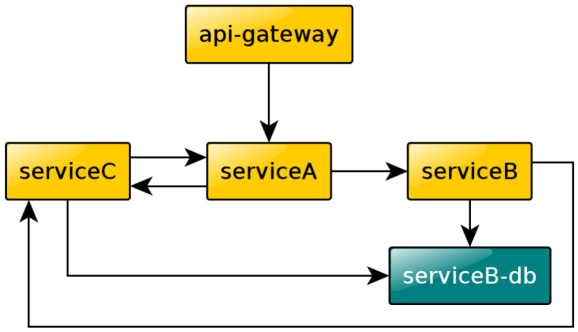


Figura 19: Grafo dello schema del Synthetic

e quando si analizza il **grafo** che era prodotto da **Aroma** si può osservare lo schema(Figura 20):

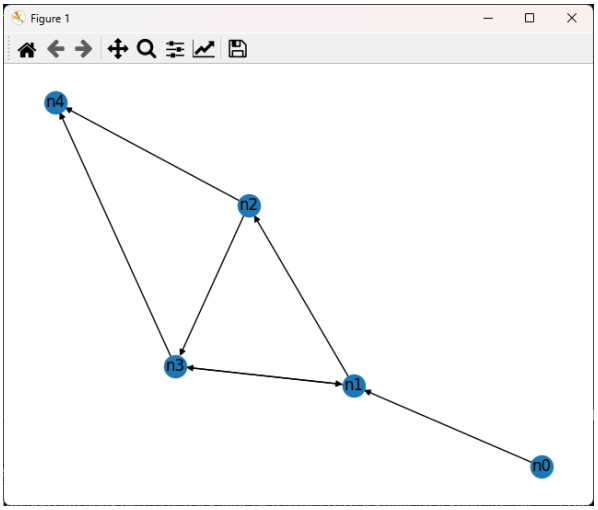


Figura 20: GraphML allo stato iniziale del Synthetic

I nodi che si possono osservare stampati nella Figura 20 sono:

- n0 = api-gateway
- n1 = serviceA
- n2 = serviceB
- n3 = serviceC
- n4 = serviceB-db

Come indicato dallo strumento AMARI, è possibile riconoscere che al momento non è presente il No API Gateway(Figura 21).

```
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistance), cd(Cyclic Dependency), e(exit))nga
API Gateway found
There is a Gateway API
Gateway API node:  api-gateway
Gateway id: n0
Nodes:  ['n0', 'n1', 'n2', 'n3', 'n4']
Edges:  [('n0', 'n1'), ('n1', 'n2'), ('n1', 'n3'), ('n2', 'n4'), ('n2', 'n3'), ('n3', 'n1'), ('n3', 'n4')]
```

Figura 21: Assenza di No API Gateway in Synthetic

Sono invece presenti:

- **Dipendenza ciclica**

```
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistance),cd(
    Cyclic Dependency), e(exit))cd
There is a Cyclic Dependency
Problematic nodes:  [('n1', 'n2'), ('n2', 'n3'), ('n3', 'n1')]
Do you want to modify the problematic nodes? (y/n)
```

Figura 22: Presenza di Dipendenza Ciclica in Synthetic

- **Shared Persistance**

```
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistence), cd(
    Cyclic Dependency), e(exit))sp
there are problematic nodes
There is a Shared Persistence
Database node:  redis
Database id: n4
Problematic nodes:  ['n2', 'n3']
Do you want to modify the problematic nodes? (y/n)
```

Figura 23: Presenza di Shared Persistence in Synthetic

Esistono varie risoluzioni plausibili per eliminare gli **smell** presenti all'interno del progetto **Synthtic**[18], per la **Dipendenza ciclica** si potrebbe(Figura 22):

- Inizialmente rompere il ciclo cancellando un interconnessione tra n2 e n3(Figura 24).
- Inserire un microservizio n5 che riproponga il servizio che richiedeva n2 a n3(Figura 25).
- Connettere n5 a n2(Figura 26).

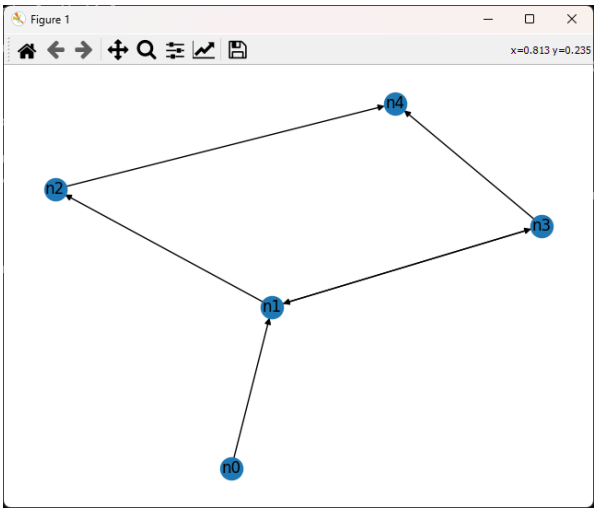


Figura 24: GraphML del Synthetic dopo la rimozione del arco n2-n3

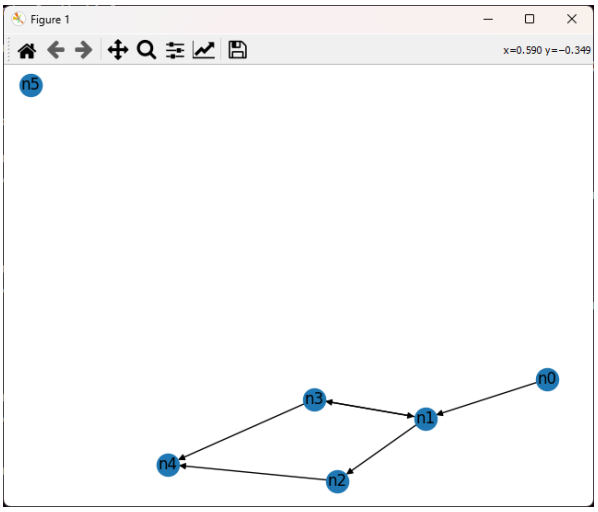


Figura 25: GraphML del Synthetic dopo aver aggiunto n5

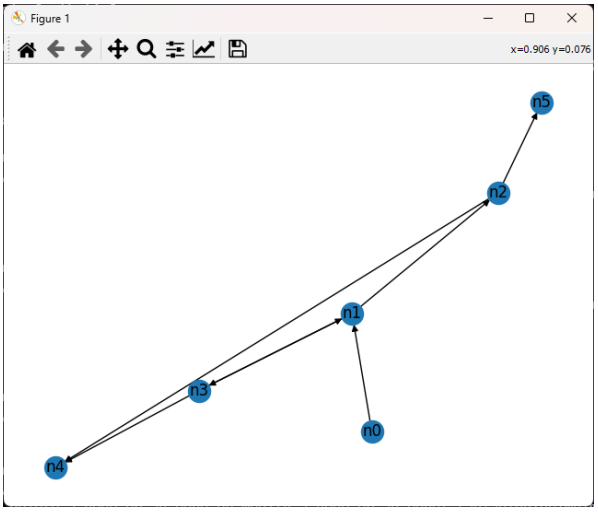


Figura 26: GraphML del Synthetic dopo l'aggiunta del arco n2-n5

Dopo queste modifiche, si osserva con lo strumento per notare se sia ancora presente la Dipendenza ciclica (Figura 27):

```
Insert the folder in which is the graphml: Synthetic
Nodes:  ['n0', 'n1', 'n2', 'n3', 'n4', 'n5']
Edges:  [('n0', 'n1'), ('n1', 'n2'), ('n1', 'n3'), ('n2', 'n4'), ('n2', 'n5'),
    ('n3', 'n1'), ('n3', 'n4')]
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistance), cd(Cyclic
    Dependency), e(exit))cd
There is a Cyclic Dependency
Problematic nodes:  [('n1', 'n3'), ('n3', 'n1')]
Do you want to modify the problematic nodes? (y/n)
```

Figura 27: Secondo controllo della presenza di Dipendenza Ciclica

AMARI segnala ancora una **Dipendenza ciclica** tra n1 e n3 per questa situazione di può creare un servizio n6 che prenda le veci del ritorno dei dati a n1.
Quindi si attuerà:

- si elimina l'arco che rientra in n1 da n3(Figura 28).
- si aggiunge un nodo n6(Figura 29).
- Si interconnettono la n3 a n6(Figura 30).

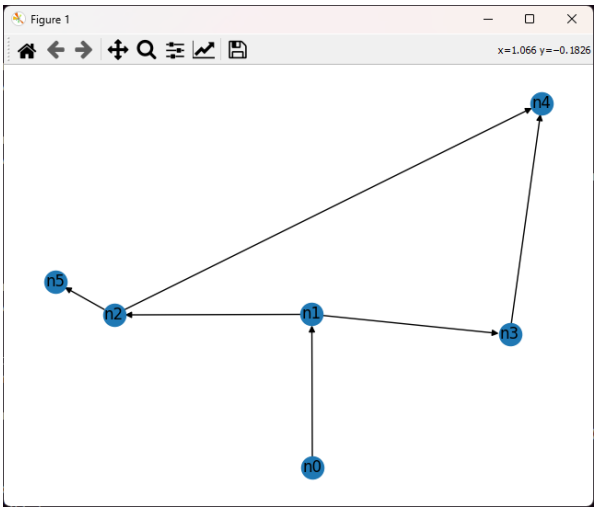


Figura 28: GraphML del Synthetic dopo la rimozione del arco n1-n3

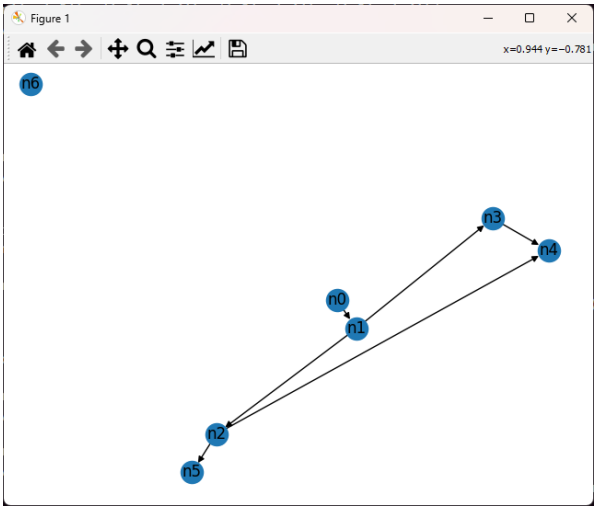


Figura 29: GraphML del Synthetic dopo aver aggiunto n6

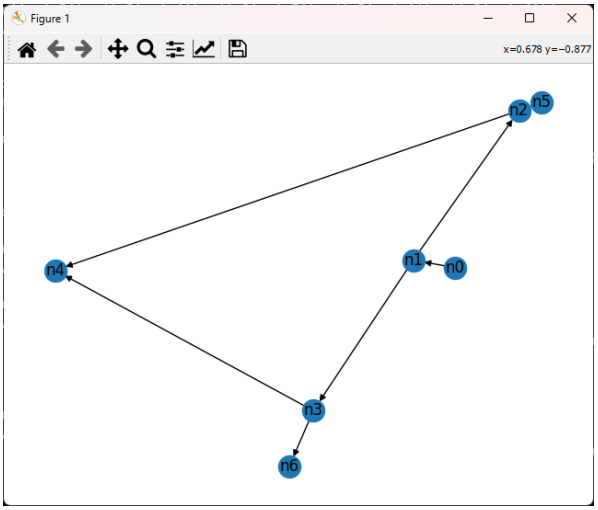


Figura 30: GraphML del Synthetic dopo l'aggiunta del arco n3-n6

Concluse queste modifiche controlliamo col tool se la **Dipendenza ciclica** sia ancora presente(Figura 31):

```
Insert the folder in which is the graphml: Synthetic
Nodes:  ['n0', 'n1', 'n2', 'n3', 'n4', 'n5', 'n6']
Edges:  [('n0', 'n1'), ('n1', 'n2'), ('n1', 'n3'), ('n2', 'n4'), ('n2', 'n5'), ('n3', 'n4'), ('n3', 'n6')]
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistance), cd(Cyclic Dependency), e(exit))cd
There is no Cyclic Dependency
```

Figura 31: Terzo controllo della presenza di Dipendenza Ciclica

Ora si devono attuare le modifiche per rimuovere la **Shared Persistance** che il tool identifica nei nodi(Figura 23):

```
Insert the folder in which is the graphml: Synthetic
Nodes:  ['n0', 'n1', 'n2', 'n3', 'n4', 'n5', 'n6']
Edges:  [('n0', 'n1'), ('n1', 'n2'), ('n1', 'n3'), ('n2', 'n4'), ('n2', 'n5'), ('n3', 'n4'), ('n3', 'n6')]
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistance), cd(Cyclic Dependency), e(exit))sp
there are problematic nodes
There is a Shared Persistence
Database node:  redis
Database id: n4
Problematic nodes:  ['n2', 'n3']
Do you want to modify the problematic nodes? (y/n)
```

Figura 32: Controllo di presenza della Shared Persistence in Synthetic

Quindi, per affrontare queste modifiche, potremmo considerare l'inserimento di un nuovo database per il nodo n2 o n3 (come mostrato nella Figura 32) che contenga una porzione del database presente in n4. In questo modo, eviteremmo di richiamare più volte gli stessi dati. Quindi aggiungendo il nodo n7 nuovo database il grafo diverrà(Figura 33).

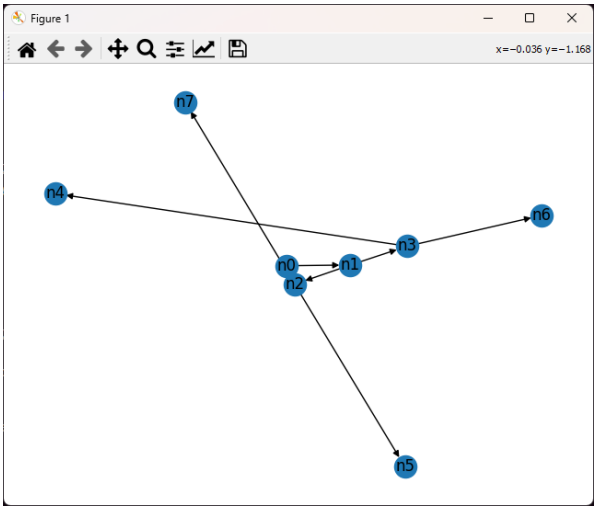


Figura 33: GraphML del Synthetic dopo l'aggiunta del arco n2 al nuovo nodo database n7

Il tool, reinserendo il grafo del progetto Synthetic, presenta:

```
Insert the folder in which is the graphml: Synthetic
Nodes:  ['n0', 'n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7']
Edges:  [('n0', 'n1'), ('n1', 'n2'), ('n1', 'n3'), ('n2', 'n5'), ('n2', 'n7'), ('n3', 'n4'), ('n3', 'n6')]
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistance), cd(Cyclic Dependency), e(exit))sp
There are no problematic nodes
There is no Shared Persistence
```

Figura 34: Assenza della presenza della Shared Persistence in Sythetic

Ora, poiché tutti gli smells sono stati eliminati, possiamo confermare di aver trovato una soluzione valida per il progetto Synthetic.

## 5.3 Progetto Online Boutique

L'ultimo progetto che abbiamo analizzato è Online Boutique[19], quindi procediamo all'estrazione dei smells presenti, seguendo la stessa procedura adottata per Synthetic. Inserendo nel tool **Online Boutique** il grafo si presenterà come(Figura 36):

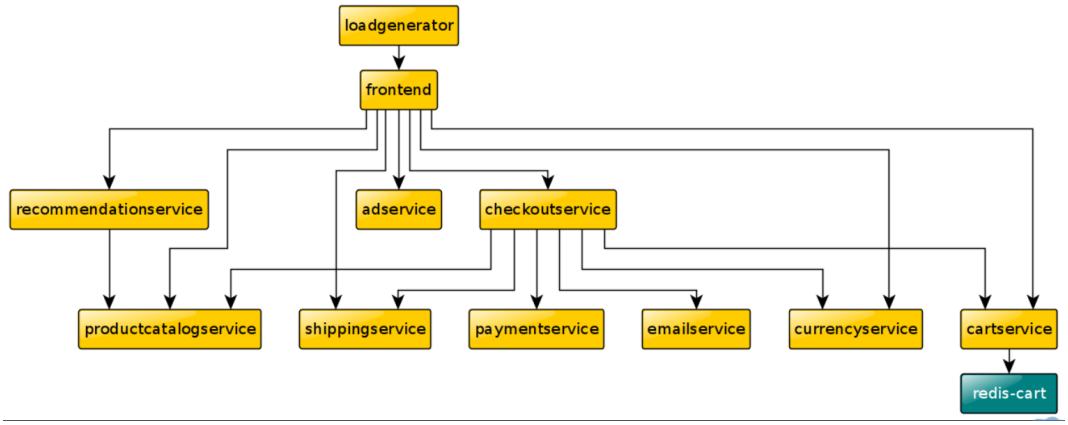


Figura 35: Grafo dello schema di Online Boutique

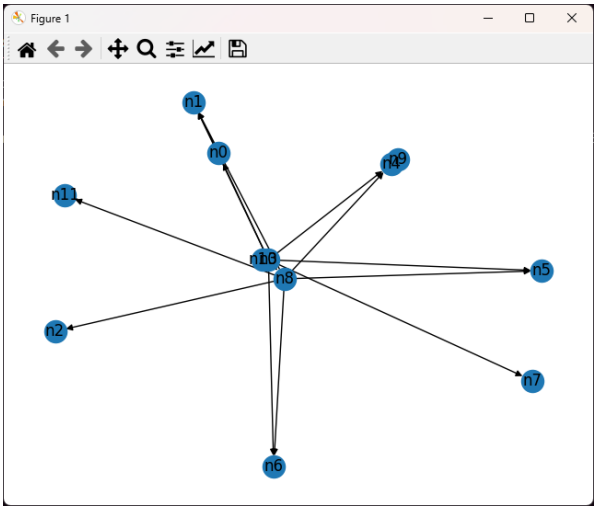


Figura 36: GraphML allo stato iniziale di Online Boutique

I nodi che si possono osservare stampati sono, come ricavato dal confronto tra la Figura 35 e la Figura 36:

- n0 = recommendationservice
- n1 = productcatalogservice
- n2 = paymentservice
- n3 = frontend
- n4 = cartservice

- n5 = shippingservice
- n6 = currencyservice
- n7 = adservice
- n8 = checkoutservice
- n9 = redis
- n10 = loadgenerator
- n11 = emailservice

Dopo l'analisi di AMARI, non sono presenti gli smell:

- **Shared Persistance**:

```
Insert the folder in which is the graphml: Online Boutique
Nodes:  ['n0', 'n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8', 'n9', 'n10', 'n11']
Edges:  [('n0', 'n1'), ('n3', 'n4'), ('n3', 'n5'), ('n3', 'n0'), ('n3', 'n1'),
        ('n3', 'n6'), ('n3', 'n7'), ('n3', 'n8'), ('n4', 'n9'), ('n8', 'n11'),
        ('n8', 'n1'), ('n8', 'n4'), ('n8', 'n5'), ('n8', 'n2'), ('n8', 'n6'),
        ('n10', 'n3')]
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistance), cd(Cyclic
    Dependency), e(exit))sp
There are no problematic nodes
There is no Shared Persistence
```

Figura 37: Assenza dello Shared Persistence in Online Boutique

- **Cyclic Dependency**:

```
Insert the folder in which is the graphml: Online Boutique
Nodes:  ['n0', 'n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8', 'n9', 'n10', 'n11']
Edges:  [('n0', 'n1'), ('n3', 'n4'), ('n3', 'n5'), ('n3', 'n0'), ('n3', 'n1'),
        ('n3', 'n6'), ('n3', 'n7'), ('n3', 'n8'), ('n4', 'n9'), ('n8', 'n11'),
        ('n8', 'n1'), ('n8', 'n4'), ('n8', 'n5'), ('n8', 'n2'), ('n8', 'n6'),
        ('n10', 'n3')]
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistance), cd(Cyclic
    Dependency), e(exit))cd
There is no Cyclic Dependency
```

Figura 38: Assenza della Dipendenza Ciclica in Online Boutique

Lo smell presente è invece il No API Gateway:

```
Insert the folder in which is the graphml: Online Boutique
Nodes:  ['n0', 'n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8', 'n9', 'n10', 'n11']
Edges:  [('n0', 'n1'), ('n3', 'n4'), ('n3', 'n5'), ('n3', 'n0'), ('n3', 'n1'),
        ('n3', 'n6'), ('n3', 'n7'), ('n3', 'n8'), ('n4', 'n9'), ('n8', 'n11'),
        ('n8', 'n1'), ('n8', 'n4'), ('n8', 'n5'), ('n8', 'n2'), ('n8', 'n6'),
        ('n10', 'n3')]
What do you want to check in the graph? (nga(No Gataway API), sp(Shared Persistance), cd(Cyclic
    Dependency), e(exit))nga
No API Gateway found
root node:  loadgenerator
root id: n10
Problematic nodes:  None
Do you want to add one or modify the graph? (y/n)
```

Figura 39: Presenza del No Gateway API in Online Boutique

In questo caso, tuttavia, non si verificheranno errori, poiché conoscendo la struttura possiamo stabilire che il "loadgenerator" (come mostrato nella Figura 39) non sia l'api-gateway della struttura. Pertanto, l'unica modifica necessaria sarà l'aggiunta di un nuovo nodo radice per rappresentare l'API.

# 6 Conclusioni e Future Estensioni

In questa tesi, è stato proposto AMARI, *Automatic Microservices Architecture Refactoring Instrument*, uno strumento nell'ambito del trattamento e della ricostruzione delle architetture basate su microservizi.
Il principale scopo di AMARI è affrontare la rimozione degli smells e rendere possibile la ricostruzione di strutture altrimenti irrecuperabili.
AMARI è stato utilizzato per condurre l'analisi di tre progetti: due Open-Source (Online Boutique[19] e SockShop[17]), ed uno sintetico[18] (generato manualmente seguendo i pattern chiave valore imposti da Zipkin V2). Durante l'analisi, AMARI ha preservato la struttura del grafo di partenza di ciascun progetto e ha generato un file Zipkin che riporta tutte le modifiche effettuate.
Nel corso dell'analisi, sono stati identificati e risolti tre microservices smells: No Api-Gateway, Shared Persistence e Cyclic Dependency. Le soluzioni adottate per affrontare questi problemi sono state descritte nella sottosezione 3.3.1 e in dettaglio nella sottosezione 5.1.
Questo approccio ha consentito l'utilizzo del tool sviluppato nella tesi con qualsiasi applicazione che richieda input e output nel formato specifico, non limitandosi ad Aroma[7], che manipolano un file JSON in input e producono un GraphML in output(Figura 40).

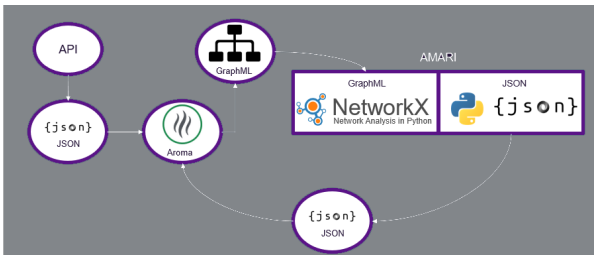


Figura 40: Schema di relazione tra Aroma e AMARI

Per concludere, esamineremo le future estensioni pianificate per il tool, analizzando le prossime fasi di sviluppo e le potenziali nuove funzionalità che potrebbero essere implementate.
Il focus principale di AMARI è offrire la capacità di rilevare e gestire vari tipi di smell architetturali, in linea con le prossime evoluzioni pianificate già per Aroma.
Alcuni esempi di questi smell architetturali potrebbero includere:

- Wrong Cuts
- Inappropriate Service Intimacy
- Mega Services
- Shared Libraries
- API Versioning
- Hard-Coded Endpoints

Considerando prospettive future, sarebbe opportuno esplorare la fattibilità di sviluppare uno strumento con funzionalità di versionamento specifiche per i file .json, in modo da generare diverse versioni per soddisfare varie esigenze d'uso.

# Riferimenti bibliografici


[1] *Cosa sono i microservizi?* URL: https://aws.amazon.com/it/microservices/.

[2] Justus Bogner, Stefan Wagner e Alfred Zimmermann. «Towards a Practical Maintainability Quality Model for Service-and Microservice-based Systems». In: set. 2017, pp. 195–198. DOI: 10.1145/3129790.3129816.

[3] *Migrazione di un'applicazione monolitica a microservizi su Google Kubernetes Engine | Cloud Architecture Center*. URL: https://cloud.google.com/architecture/migrating-a-monolithic-app-to-microservices-gke?hl=it.

[4] *Google Kubernetes Engine (GKE)*. URL: https://cloud.google.com/kubernetes-engine?hl=it.

[5] *Che cos'è Istio?* URL: https://cloud.google.com/learn/what-is-istio?hl=it.

[6] *Gestione delle API Apigee*. URL: https://cloud.google.com/apigee?hl=it.

[7] Paolo Bacchiega, Ilaria Pigazzini e Francesca Arcelli Fontana. «Microservices smell detection through dynamic analysis». In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2022, pp. 290–293. DOI: 10.1109/SEAA56994.2022.00052.

[8] Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

[9] Isuru Udara e Indika Perera. «Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach». In: lug. 2021. DOI: 10.1109/MERCon52712.2021.9525743.

[10] *OpenZipkin A distributed tracing system*. URL: https://zipkin.io/.

[11] *Jaeger: open source, end-to-end distributed tracing*. URL: https://www.jaegertracing.io/.

[12] *yEd Graph Editor*. URL: https://www.yworks.com/products/yed.

[13] *The GraphML File Format*. URL: http://graphml.graphdrawing.org/index.html.

[14] *NetworkX — NetworkX documentation*. URL: https://networkx.org/.

[15] *Che cos'è JSON?* URL: https://www.oracle.com/it/database/what-is-json/.

[16] *Swagger UI, Zipkin API*. URL: https://zipkin.io/zipkin-api/.

[17] *Sock Shop : A Microservice Demo Application*. URL: https://github.com/microservices-demo/microservices-demo.

[18] *analyzedProjects/Synthetic · master · essere.lab.public / Aroma · GitLab*. URL: https://gitlab.com/essere.lab.public/aroma/-/tree/master/analyzedProjects/Synthetic?ref_type=heads.

[19] *GoogleCloudPlatform/microservices-demo*. URL: https://github.com/GoogleCloudPlatform/microservices-demo.