USING DEPENDENCY GRAPH AND GRAPH THEORY CONCEPTS TO IDENTIFY ANTI-PATTERNS IN A MICROSERVICES SYSTEM: A TOOL-BASED APPROACH

Isuru Udara Piyadigama Gamage

189316U

Degree of Master of Science in Computer Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

April 2021

USING DEPENDENCY GRAPH AND GRAPH THEORY CONCEPTS TO IDENTIFY ANTI-PATTERNS IN A MICROSERVICES SYSTEM: A TOOL-BASED APPROACH

Isuru Udara Piyadigama Gamage

189316U

Thesis/Dissertation submitted in partial fulfilment of the requirements for the degree Master of Science in Computer Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

April 2021

DECLARATION

I declare that this is my own work and this thesis/dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis/dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

…….. ……

I.U. Piyadigama Gamage Date

The above candidate has carried out research for the Masters thesis/ Dissertation under my supervision.

UOM Verified Signature

…….. ……

Dr. Indika Perera (Research Supervisor) Date

ABSTRACT

Microservice based application developments are becoming more and more popular and becoming the common trend in implementing large scale applications. Unlike the traditional monolith applications, microservice applications are composed of many services hence there is an immense possibility of anti-patterns introduced in to the system. To identify these design problems, detailed analysis of the architecture needs to be performed. Tracing data plays a significant role here but it is less useful on its own because of the complexity and the increased quantity of information. Diving in to this data and analysing them to identify anti-patterns exist is a burdensome task because of the lack of tools available thus it requires an immense human intervention and effort due to its complexity. The most common tools available for this kind of scenarios are focused on presenting the information visually so that they are more human readable. The dependency graph of a microservices system, generated by these tools is a good example. However, these tools are not up to performing analysis on the traced data and presenting developers with more statistical information such as metrics along with visualization techniques so that developers can take actions by evaluating metrics and visualizations both, not just by the visualizations. In this thesis we present a solution, a tool for this problem which is capable of utilizing traced data of a microservice system to generate dependency graph and thereby extract metrics using graph theory concepts and algorithms from the dependency graph. The graph theory concepts such as Degree, Clustering coefficient, Paths and distances are used to generate relevant metrices. These metrices are presented to the user in a statistical and a visualized way by the proposed solution so that developers can evaluate them and take decisions on refactoring. To build the proposed tool, we used some already available, industry well recognized tools to develop relevant applications, trace the data and store the data. We use a system which has been developed with microservice style for the analysis. The system has been introduced with anti-patterns intentionally. In this research we focus on anti-patterns specifically The Knot, Nano service, Chained service, Bottleneck service and Cyclic dependency. We analyse the microservice system we have developed with the solution tool presented to identify these anti-patterns.

Keywords

Microservices, Distributed systems, Anti patterns, Graph theory, Dependency graph, Tracing

ACKNOWLEDGEMENTS

I would like to specially thank my supervisor Dr. Indika Perera for the immense support and guidance given throughout the research. He conducted the lectures for core modules related to Software Architecture that were immensely useful in this research. Then I would like to thank Dr. Sanath Jayasena who conducted the lectures for Advanced Algorithm module which the knowledge gained was also extremely helpful in this research.

My gratitude towards all the lecturers for all the modules I took during the course that equipped me with basic to advanced knowledge in Computer Science. I am grateful to Dr. Charith Chitraranjan, Mrs. Vishaka Nanayakkara and Dr. Malaka Walpola who accepted me for this MSc course in the selection interview.

I would like to thank both technical and non-academic staff of the department for their kind and prompt support which made the process smooth. I would like to thank my dear colleagues in the MSc course that supported me throughout the course.

Finally, I wish to thank my parents for all the support given throughout the course to complete it successfully.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

LIST OF ABBREVIATIONS

API	Application Programming Interface
IOT	Internet of Things
SIC	Service Invocation Chain
REST	Representational State Transfer
IIS	Internet Information Services
HTTP	Hypertext Transfer Protocol
SUT	System Under Test
QOS	Quality of Service
SOA	Service Oriented Architecture
OOP	Object Oriented Programming

1. INTRODUCTION

1.1 Background

In modern software development world “microservices” has become a very hot topic because of the benefits of adapting it over traditional approaches. The traditional approach of monolithic application development has obvious drawbacks when it comes to performance, scalability, maintainability[31]. This led to an immergence of a new architectural pattern in recent years where the application is decoupled in to fine-grained small components that encapsulates its own business capabilities. These small modules are known as “Microservices” and have become the mainstream architectural pattern in large-scale software development [1]. These fine-grained services run separately in their own process and communicate through message passing [30]. In a microservice system each service complies with single responsibility principle [35] and the architectural style inherently encourages agile software development [31].

The characteristic of each microservice should comply with single responsibility principle emphasize on how a service should be defined, how the boundaries with other services are drawn and when the system extends or changes where the code changes should go. This makes the microservices loosely coupled by nature [36]. As a result, developers can independently change services without effecting other services in the system. The loosely couple nature allows microservices to be scaled and deployed independently [7].

All the above qualities encourage companies to adapt microservice architectural style for their systems. The downside is if the services are not properly designed then it will lead to a tight coupling between services and hinder most or all of the above mentioned qualities and benefits which can be gained by adapting the architectural style [35]. The things will get worse with the unavoidable complexity of the architecture. A microservice application may contains hundreds or thousands of services [32]. The services tend to be changed and released frequently and the distributed manner of the architecture increases the complexity [33]. This makes the microservice system vulnerable to anomalies and

failures hence monitoring at runtime is essential in a microservice system but also the architecture of the system must be designed properly [31].

Anti-patterns are adversely affecting common solutions to recurring problems which is the opposite of design-patterns that are well recognized commonly used good solutions to recurring problems that are faced by architects and developers when designing and implementing a software system [37]. A microservice architecture evolves frequently when introducing new functionalities in terms of functionalities and quality of service (QoS) and this nature can lead to poorly designed services which can often result in appearing anti-patterns in the system. These anti-patterns affect the design of the system adversely such that the quality factors such as maintainability are degraded [37]. Later the complexity of the architecture makes the anti-patterns detection a more challenging task [2].

Microservice system are allowed and supposed to change rapidly and efficiently. The maintainability [34] is a key quality attribute for this kind of continuously changing systems. It is immensely important for an organization which design, implement and maintain microservice based systems to have metrics to evaluate maintainability of the system quantifiably besides the systematic understanding of maintainability [33]. The anti-patterns exist in a microservice system directly affects the quality attribute maintainability. Existing work is more focused on presenting approaches to measure quality attributes such as maintainability considering the design properties such as coupling, cohesion, service granularity etc. using some metric model [38]. But there is a lack of approaches that visually and quantifiably identify anti-patterns in a microservice system.

Most approaches generate service dependency graph of the microservice system and require developer to manually put effort to analyse it [4]. When the microservice system grow in scale this becomes a burdensome task. Dependency graph generation can be done as static and dynamic means and existing work mostly focused on static means [40]. This

also requires manual effort from developers to capture the necessary data in order to generate the service dependency graph.

There are very few approaches that utilize graph theory concepts and algorithms to evaluate and identify anti-patterns in a microservice system [5]. In this research we emphasize the potential of adapting graph theory concepts and algorithms to generate relevant metrics to evaluate a dependency graph of a microservice system and thereby identifying anti-patterns in the system.

1.2 Problem Statement

There are tools developed to visualize microservice systems architecture in a more human readable way as a dependency graph. But they are not capable of giving any statistical insights in to the architecture [4]. Developers need to put manual effort to analyse these dependency graphs to identify anti-patterns exist in the system which is time consuming and it is a difficult task when it comes to large microservice systems because of the complexity and the increased quantity of information to analyse.

There are tools and techniques available to gather data related to microservice systems such as details of services in a microservice system, request paths between services, number of requests per time unit etc. These data are used to detect failures in a system, to generate and monitor performance metrics related to CPU usage and hard drive usage, request and response time in a distributed system etc. Distributed tracing tools are heavily used to gather these data [3].

The problem is, there are not many tools that process these traced data and give insights in to the system architecture in a more statistical way so that developers can identify anti patterns in a microservice system and can take appropriate refactoring actions.

1.3 Motivation

Developers are in a challenging situation when they are to investigate problems in a microservice system. Sometimes service quality problems are reported by clients. This could be due to infrastructure problems or due to poorly designed architecture. Sometimes development teams get already developed large microservice systems and they are in a situation to review the existing architecture before proceeding further. To assess the architecture quality, some kind of efficient mechanisms should be in place and in that situation the architecture diagram in hand is less useful.

The architecture diagram may have hundreds of services all over with unclear dependencies and the diagram could be outdated as well. Later on, they use some kind of a tracing tool to trace the runtime data of the system to generate the dependency graph to visualize the services and their dependencies. Then they use the dependency graph to analyse the problematic components of the architecture. Even though now developers have something more readable in their hand, the analysis remains difficult as they have to put manual effort to analyse the dependency graph.

Some developments teams have a very less or no knowledge regarding microservice application development practices. If they are to build a new microservice system or migrate an existing monolithic application to microservice architecture, they will have to constantly worry about whether system is being up to best standards.

The architectural anti-patterns are possible in a microservice system and most of the time in poorly designed systems. The analysis and identification process of the problematic components must be simplified.

Introducing a more statistics driven and human readable technique for the developers to easily identify anti-patterns exist in a microservice system is the prime intention behind this work. The analysis of large microservice systems is a burdensome task because of the complex nature of the system. The raw tracing data on its own is less useful and complex to utilize and get insights in to the architecture.

1.4 Objectives

In this research we propose a tool-based approach which applies graph theory concepts on a microservice dependency graph to generate relevant metrics to analyse and identify anti-patterns. The dependency graph is generated using runtime data dynamically. Our prime intention is to present a solution to identify anti-patterns exist in a microservice system with minimal effort.

As the first step we gather the data related to microservice application including services and their communication links. We don't assume having an up-to-date architecture diagram in hand and manually creating the data model of the architecture. So, we gather these data at the system productive time (run time) using a distributed tracing mechanism. These data are known as trace data in distributed tracing domain and trace data of a system contains critical information such as request execution paths, number of requests per time unit, services encompass the system and their inter-relationships [5]. Microservices typically communicate over a light weight messaging protocol such as REST or a queuing mechanism [7]. For this research we consider only synchronous REST based communication between services.

Then we map the trace data in to graph models considering the services as nodes, request paths and directions of the requests as edges, information of services as node properties and information of requests as edge properties.

There can be hundreds of services in a microservices system hence the graph models we generate may have hundreds of nodes and edges. So, we persist the graph model in a graph database in order to make the data retrieval process efficient. We make use of the graph algorithm libraries provided by the graph database when generating metrics.

We generate the dependency graph of the system and the relevant metrics using the persisted data model. The generated metrics are based on graph theory concepts such as Degree, Clustering coefficient and Path distances [17]. The dependency graph and other relevant graph models are provided as visualizations.

Using these visualizations and metric values we get insights in to the system architecture in a more statistical way so that the developers can identify design problems and justify a refactoring of a particular component in a more quantifiable way.

1.5 Thesis Structure

The outline of the report is as follows. Chapter two is focused on literature review and there we initially discuss important concepts which are relevant to the research area such as Microservices, Anti-patterns, Distributed tracing, Service dependency graph and Graph theory concepts. Then we discuss the related work in this research area under 3 categories; service based metrics, service dependency graph and anti-patterns in SOA. Chapter three contains the methodology which explains how we approach the solution. We discuss about the data extraction and matric evaluation of the proposed solution. Chapter four explains the implementation details of the proposed tools. In chapter five, the sample microservice application we developed is evaluated using the tool developed. Finally, in chapter six we conclude the research and discuss about future work.

2 LITERATURE REVIEW

In this chapter, first we discuss about the important concepts related to the research area. The purpose is to equip the readers with sufficient knowledge regarding the core concepts to proceed. Then we discuss the related work in the research area.

2.1 Concepts

In this section we explain the core concepts related to the research area namely Microservices, Anti-patterns, Distributed tracing, Service dependency graph and Graph theory concepts.

2.1.1 Microservices

Recent software engineering evolvements and improvements pave the path to new software development styles. Few of such improvements are distributed cloud computing facilities, web service improvements such as Application Programming Interfaces (API) and recent emergence of containerized application deployments [6].

Microservices is an architectural style which composes an application as a combination of many fine-grained services. Unlike a traditional monolithic application which encapsulates all the application capabilities in to one process, a microservice system architecture is loosely coupled. Each service in a microservice system has a well-defined bounded context and encapsulates relevant fine-grained business capabilities. This well-defined bounded context facilitates high cohesiveness because all the business logics encompass the bounded context are in a single service. Because of the low coupling nature of the services, they are independently deployable. The maintainability and the testability of the services are improved because each service can be tested separately

without depending on the other services. Low coupling improves the availability of services and the services can be scaled out individually to serve the load [7].

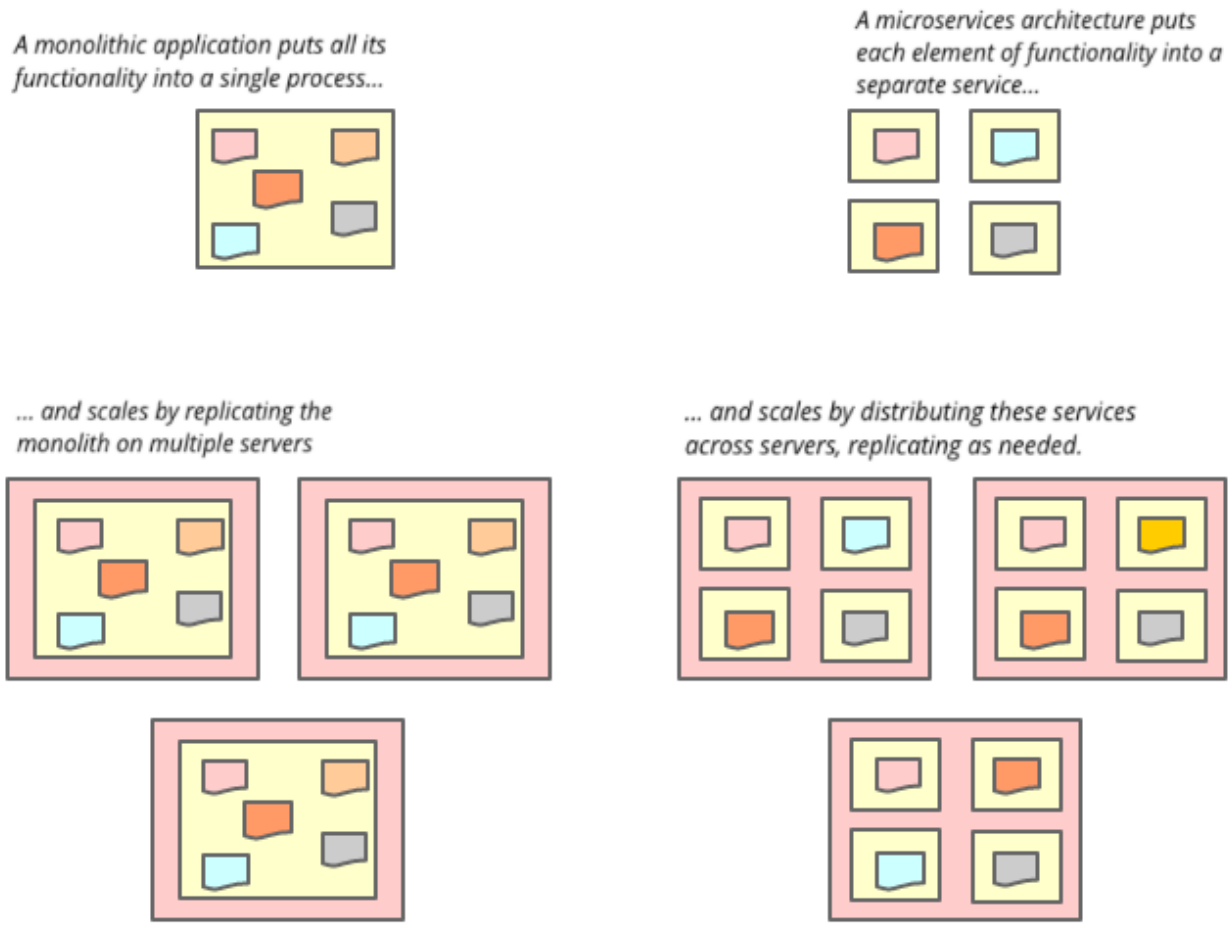


Figure 1: Monolithic and Microservice architectural styles [7]

Figure 1 illustrate both monolithic and microservice architectural styles and explains the how the application components are organized and how they are scaled in the servers.

2.1.2 Anti Patterns

The obvious advantages of having a microservices architecture, persuade companies to convert their monolithic applications to microservice applications. However, this is not an easy task and if not done carefully, the end result would be a less maintainable microservice system with performance issues. Companies and developers generally start the migration process without any prior experience or knowledge in microservices. There are only few cases where the migration process was supported and consulted by hired consultants. Therefore, companies encounter problems during implementation and after implementing the system due to lack of knowledge regarding bad practices and anti-patterns [8].

There are common migration problems identified [9]:

- The non-critical aspects of the monolithic application are focused first and starting them first.
- The migration of legacy database is not planned and the microservice system is built by connecting it to the same legacy database. The recommended way is, each microservice should have its own database hence existing database must be split carefully.
- Microservices communicate with each other over the network. This adds complexity to the architecture and the dependencies between microservices need to be carefully designed otherwise performance related problems may occur.

The anti-patterns exist in a microservice system are a result of these problems that has not been properly addressed in the migration process. In this research we focus on five anti-patterns that effect a microservice system adversely.

The Knot

This anti-pattern is a result of splitting a monolithic application as it is in to small services in the migration process. That is a wrong way of converting a monolithic system in to a microservice system. This split makes the services low cohesive hence tightly coupled. This tightly coupled behaviour makes reusability of a service, very limited. This also compromises one of the most important benefit and a quality of a microservice system, independent deployment of services. This low cohesive tightly coupled architecture prevents independent deployment of services. This kind of architecture is complex in nature and results in low availability and high response time of services [10][11].

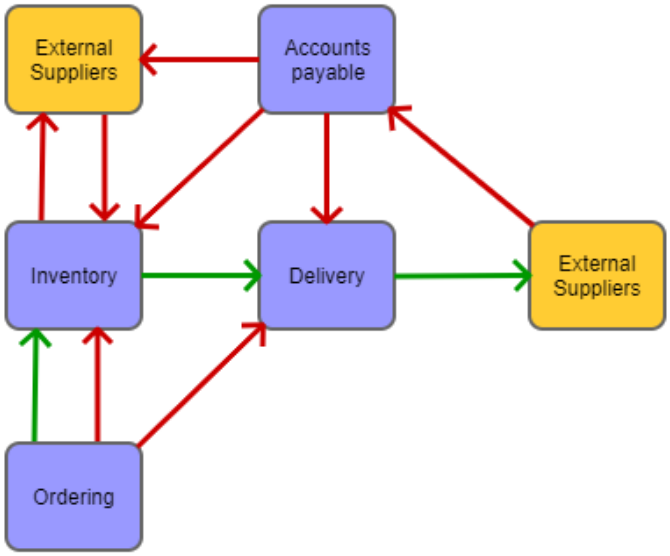


Figure 2: The Knot in a microservice system

Nano service

This is also known as Tiny service anti-pattern. In microservice architecture, a single service must encapsulate a complete business capability. This makes the service loosely coupled from other services and will facilitate consumers to get all the corresponding business-related data from that service itself without making consumers to access other services to fulfil the requirement. But a nano service is a small service which has not been built around a business capability. It just contains few operations and requires multiple coupled services to fulfil a consumer request. This makes the development of the system, complex and reduce the re-usability as well [10][11].

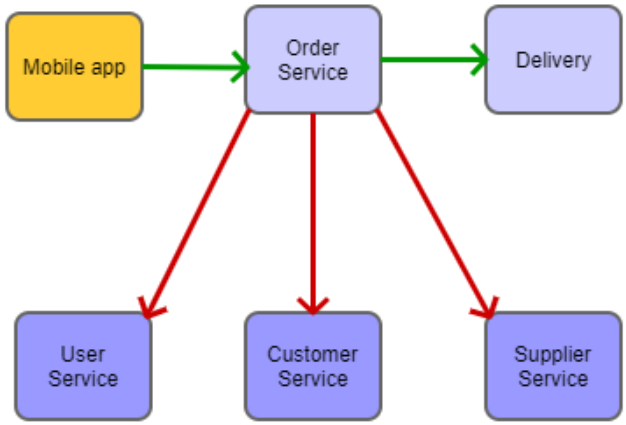


Figure 3: A microservice system with three nano services

Service chain

In this anti-pattern, chain of multiple services is invoked to serve a single user request. A chain of services is reserved to satisfy a single functionality. This also exhibit the low cohesion between services hence high coupling. A change in one service might requires changes in other services hence independent deployment of services are compromised [10][11].

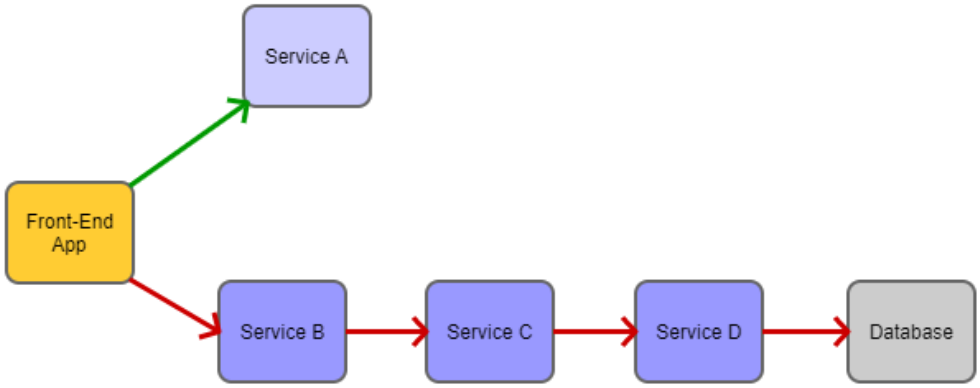


Figure 4: A microservice system with a service chain

Bottleneck service

This is also known as Mega service anti-pattern. A bottleneck service has many clients or services using it hence there is always a huge traffic in to the service and out of the service. Due to the large number of external consumers accessing the service, response time is high and it will make the service availability low for some other consumers [11][12].

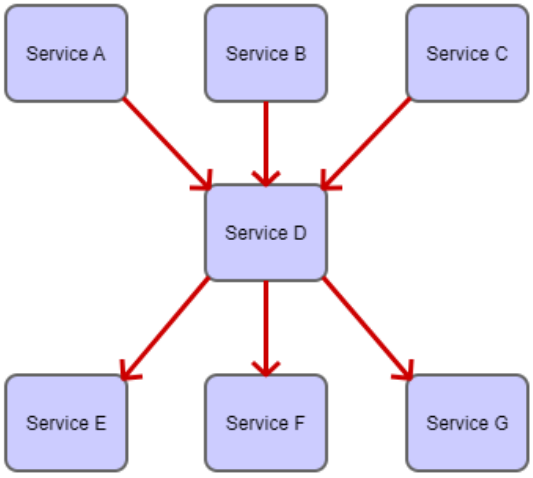


Figure 5: A microservice system with a bottleneck service

Cyclic dependency

Cyclic dependencies are a possible deign problem in microservice systems [11]. This problem could lead to a competition for resources. This also inhibit the testability and reusability of services because of the strong tight coupling between services. Services involved in the cycle are more difficult to test because they cannot be tested individually. If we make a small change to a service it could cause a ripple effect to other services and because of that the system as whole prone to bugs and crashes [5].

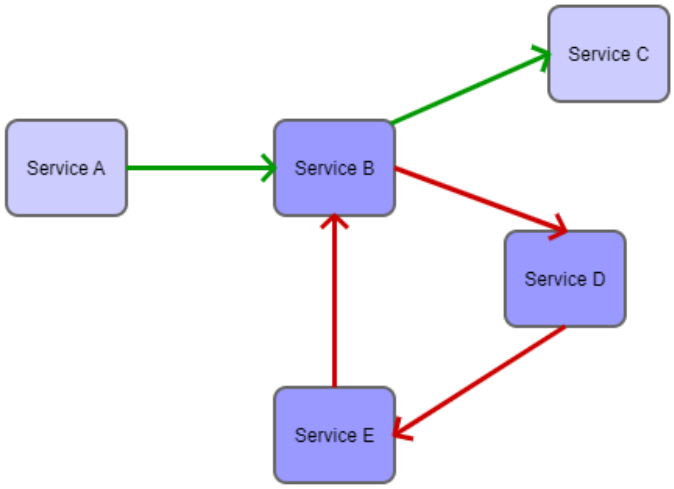


Figure 6: A possible cyclic dependency in a microservice system

2.1.3 Distributed Tracing

The emergence of large-scale service oriented and IOT applications, brought up the need of having tools for monitoring and controlling as it was a difficult task. In a traditional monolithic application, it is more than enough to have a single log file in order to understand the system status. The single log file gives the insights in to debug information, warnings and errors of the system. If this mechanism is followed in a distributed system, developers have to put a huge effort to identify the same information. Now the log files are distributed in multiple servers and developers have to evaluate all of them. Further the execution sequence of the services needs be identified. For an example by matching timestamps of the service calls. Asynchronous calls make things more complicated as they can only be evaluated on a beginning timestamp.

Different solutions and approaches immerged to tackle this problem of tracing distributed systems over time and Dapper [14] is among them. Dapper is the production distributed systems tracing platform of Google. The Google search engine is a massive distributed system and sometimes to serve a single search request, the request needs to go through hundreds of distributed services. This brings up the need of tracking how these requests navigate through this massive distributed system.

Google focused on two main characteristics for such kind of a distributed system when designing Dapper.

- Firstly, ubiquitous deployment. This is important because even if a tiniest part of the distributed system is not monitored, it will have a massive impact on the usefulness of the tracing platform.
- Secondly, continuous monitoring. The monitoring should always be turned on because it is difficult to reproduce anomalies occur in the system and continuous monitoring is a must to have characteristic.

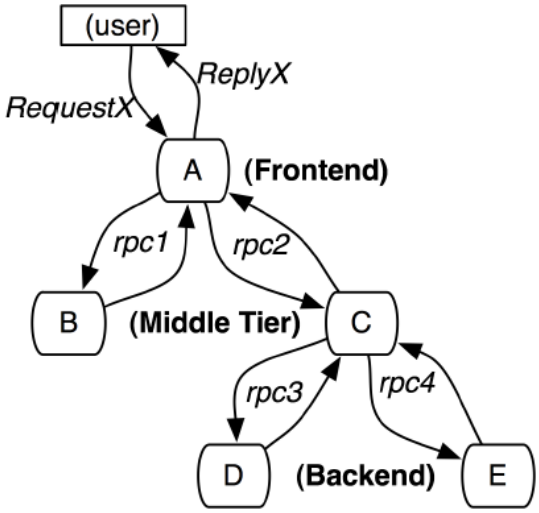


Figure 7: Communication sequence of a microservice application [14]

Figure 7 illustrates a user request navigating through a microservice based system. If all the services are monitored by a distributed tracing platform, each service will inform its own call to the tracing server as a span. Span represents a complete single service call corresponding to a single microservice such that client sends the request, server receives the request, server sends back the response and client receives the response.

Each span holds three identifiers, Trace ID, Span ID, Parent Span ID and exact timings of beginning of the request call to end. Each user request is assigned a Trace ID and the complete execution path can be represented by this. Span ID represents the ID of a particular span and Parent Span ID is only presented on child spans and is not presented on the root span.

The complete call sequence of a user request can be constructed using Trace ID, Parent Span ID and obtained timings as illustrated in figure 8.

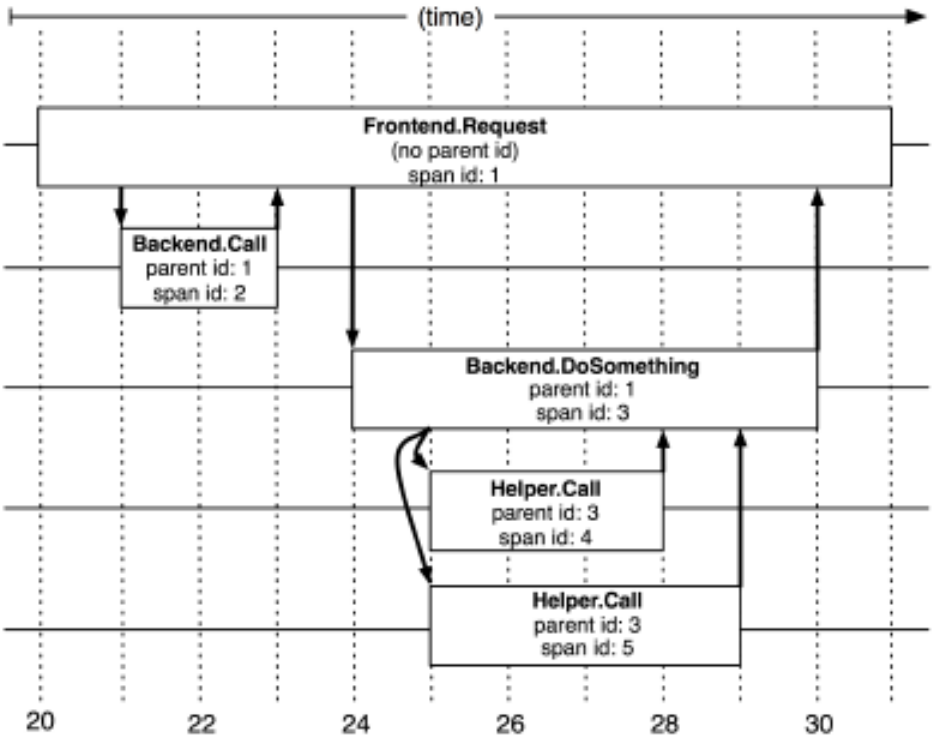


Figure 8: Complete call sequence of a user request [14]

2.1.4 Service Dependency Graph

Service Dependency graph plays an important role in monitoring and debugging large scale distributed systems. Dependency graphs are generated from the traces obtained from distributed tracing. To familiarize with the concept of Service dependency graph, we will first discuss what is a graph and its characteristics.

"A graph is set of vertices and a collection of directed edges that each connects an ordered pair of vertices" [15].

A graph is represented by its vertices/nodes and edges. If V represents vertices and E represents edges, then graph 'G' can be denoted as G = (V, E).

To introduce the concepts Vertex, Edge and Properties:

- Vertex/ Node: These are the entities in the graph. These entities represent a specific role in the domain and can be labelled accordingly. Vertices hold attributes or properties.
- Edge: This represents a connection between two vertices.
- Property: These are metadata attached to a vertex or edge.

There are mainly three types of graphs, Undirected Graph, Directed Graph and Multi-Directed Graph.

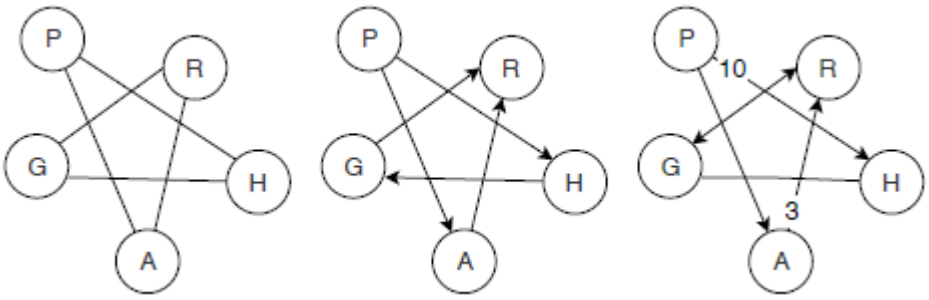


Figure 9: Types of Graphs

All the three graphs illustrated in Figure 9 have five nodes. But they are different because they have their own type. The first one is an Undirected graph because the edges of the graph do not have directions. The second graph is a Directed graph because its edges have a direction but one way. The last one is a Multi-directed graph because its edges are directed to both ways.

In the Mult-directed graph we can notice there are numbers attached to edges. These are annotations and can be presented in any type of graph we previously discussed as well. This can represent some information of any connected two vertices. For an instance, in a microservice system context if this graph represents the system such that services are represented by vertices and invocations between services are represented by edges, then an annotation value on an edge between hypothetical services A and R will represent the number of calls between services for the edge direction. In this scenario, the graph shows there are three requests performed from A to R.

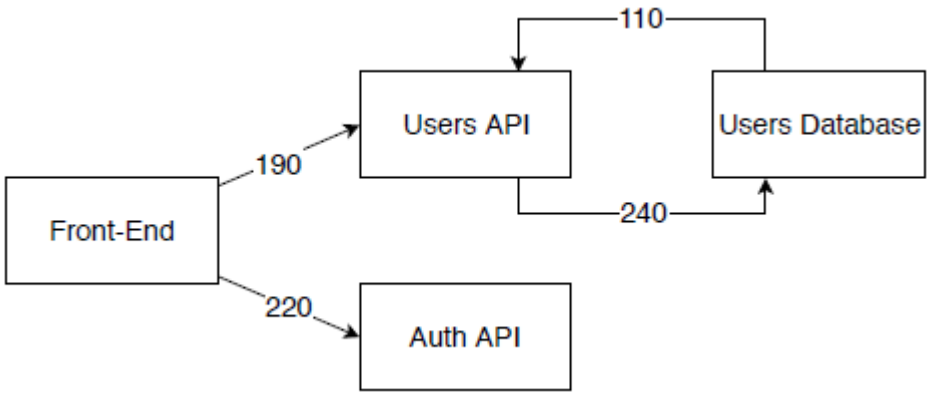


Figure 10: Microservice dependency graph

Figure 10 illustrates a service dependency graph of a microservice system. Service dependency graphs are of Multi-directed graph type because there are edges between nodes which are directed to both directions. In the diagram there are two APIs, one front-end application and one database represent in boxes (nodes). The edges represent number

of innovations between nodes. As an example, Front-end application calls User API 190 times. This dependency graph provides the state of the system in a given time interval. In the next time interval these services might disappear and new services might appear.

2.1.5 Graph Theory Concepts

Graph theory is used to analyse structural models in Mathematics. The journey of Graph theory was started from the Koinsberg bridge problem in 1735 and has been adapted in heterogeneous fields for the betterment of those fields [16].

Computer science is one of the top fields which has been utilizing Graph theoretical ideas for various aspects such as networking, data mining, clustering etc [16]. For an example, a network topology can be mapped as a graph with vertices and edges and the graph can be analysed using graph concepts.

In graph theory there are important concepts which are used to analyse a graph such as Degree, Connectedness, Adjacency Metrix, Sparse networks, Weighted networks, Bipartite networks, Paths and Distances, Connectedness, Clustering coefficient [17]. A microservice system can also be mapped to a dependency graph using traced data as we discussed in subsection 2.1.4: Service dependency graph. This graph can be analysed using graph theory concepts and, in this research, we use some of the graph concepts to analyse the dependency graph to identify design problems in a system.

Degree

Degree represents the number of edges a node has to other nodes. It is a very important property of a node. The Degree of the i^{th} node in a graph is denoted by k_i. For an undirected graph the total number of edges, L, can be formulated as the sum of the degrees of all the nodes [17]:

$$L = \frac{1}{2}\sum_{i=0}^{n} k_i$$

Equation 1: Total Degree of a Graph

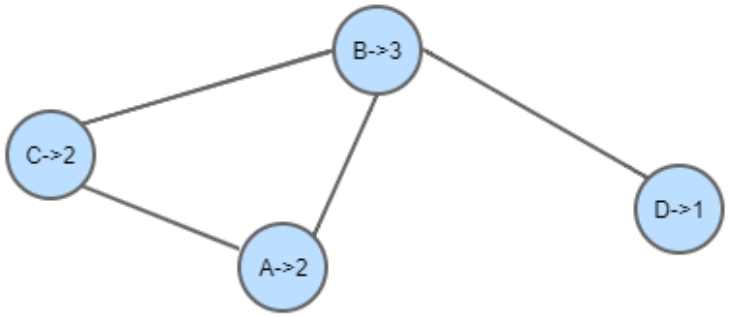


Figure 11: Associated Degrees of nodes in a graph

In a directed graph the degree of a node k_i is expressed as the total of incoming edges to the node from other nodes, k_i^{in} and outgoing edges from the node to the other nodes, k_i^{out} :

$$k_i = k_i^{in} + k_i^{out}$$

Equation 2: Degree of a Graph node

In the subsection 2.1.4: Service dependency graph, we discussed that microservice dependency graph is a directed graph. For a particular service, there could be incoming requests from other services and outgoing request to other services.

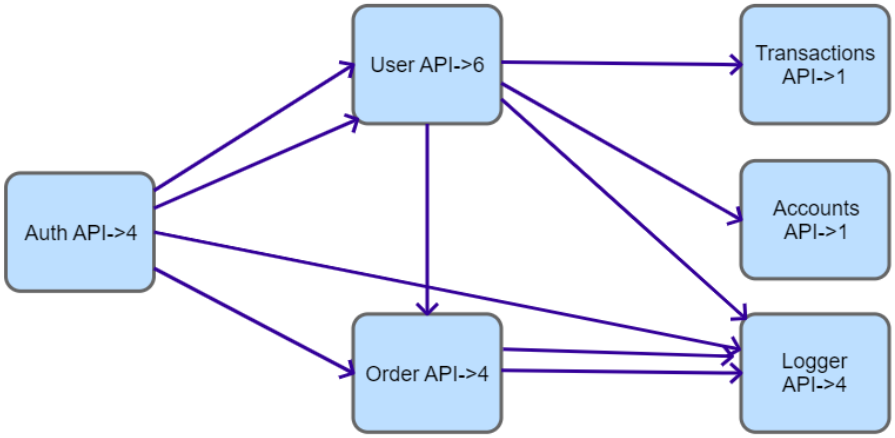


Figure 12: A microservice dependency graph with degrees

Figure 12 illustrates the dependency graph of a microservice system. In this microservice system, User service is invoked by Auth service for two connections and User service invoke Transaction service, Accounts service and Logger service. Here we ignore the fact that whether this is a single trace or more than one trace.

For the node User service, using the formula 2, we can calculate the degree k_p as below:

$k_p^{in} = 2, k_p^{out} = 4$

$k_p = 2 + 4 = 6$

Paths and Distances

In a graph, distance from one node to another is denoted by *path length*. *Path* is the route that runs along the edges from one node to another. *Length* of that *path* is the number of edges exist in the path. For given two nodes i_0 and i_n, the path P between i_0 and i_n is an ordered list of n edges [17]:

$$P = \{(i_0, i_1), (i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)\}$$

The path length is n.

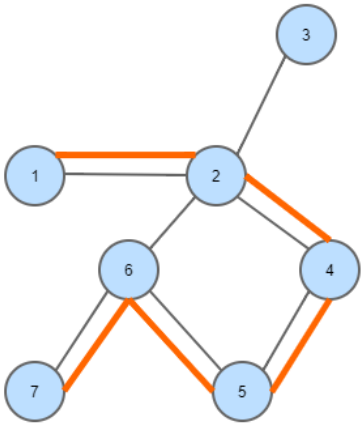


Figure 13: Path in a graph highlighted

The path illustrated in the Figure 13 from node 1 to node 7 follows the route 1 -> 2 -> 4 -> 5 -> 6 -> 7 hence the path length is 5.

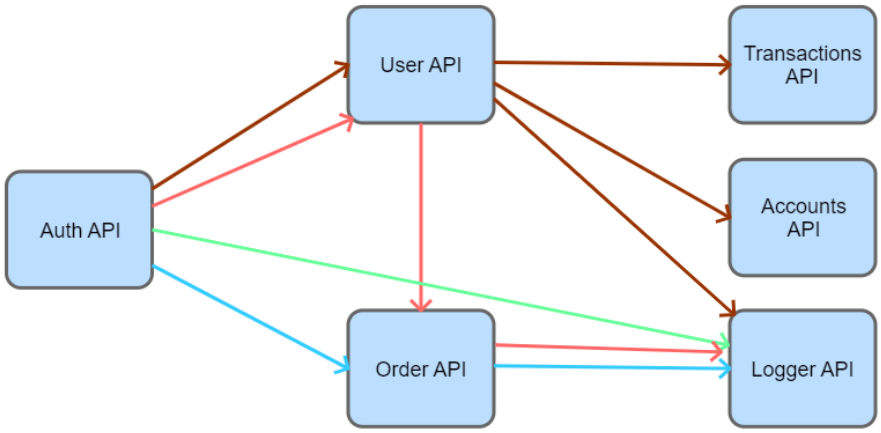


Figure 14: A microservice dependency graph with trace paths highlighted

Figure 14 illustrates the dependency graph of a microservice system with individual traces highlighted in different colours. If we consider the orange-coloured trace, Auth service invokes User service, User service invokes Order service and Order service invokes Logger service to fulfil a single user request. So, this request follows the path Auth service -> User service -> Order service -> Logger service which covers 3 service invocations and according to graph theory we can say the *length* of the *path* is 3.

- Cyclic path

  If a path of a graph starts from a node and ends with the same node then it forms a cycle hence it is called a *cyclic path*. In microservice systems sometimes we can find this kind of cycles exist in service innovation chains which is a design problem.

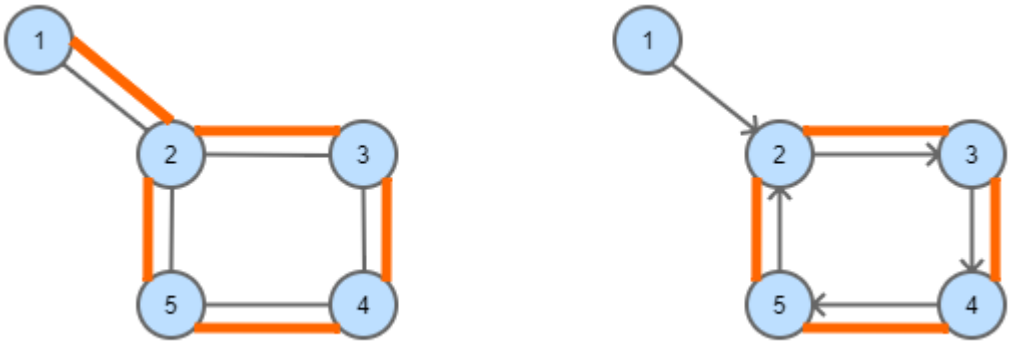


Figure 15: A Non-cyclic and a Cyclic path highlighted in a graph

Figure 15 illustrates two graphs; first graph follows a path 1 -> 2 -> 3 -> 4 -> 5 -> 2 and the second graph follows a path 2 -> 3 -> 4 -> 5 -> 2. The path in the first graph is not a cyclic path because the starting node and the ending nodes of the path are different. But in the second graph the starting node and the ending node are the same hence the path is a cyclic path.

Weighted Graphs

A system or a structurally arranged object model can have some information regarding the links or communication path between two nodes. As an example, in a mobile network this information can be the number of minutes two individuals has been talking to each other. In a graph each edge can carry this information as a *weight* attached to it hence the graph becomes a *weighted graph*. These weights are very important for analysing a graph [17].

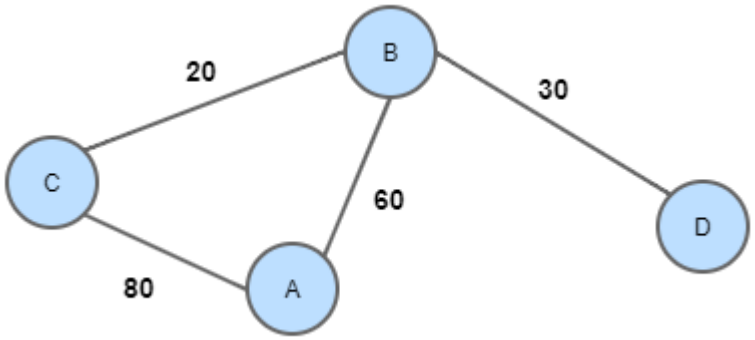


Figure 16: Weighted graph

In subsection 2.1.4: Figure 10 illustrates a dependency graph of a microservice system where each edge is annotated with number of invocations between two services. User service invokes User database 110 times and User database has a trigger which invokes User service 240 times. This information, corresponding to edges are called *weights* hence the graph is a *weighted graph*.

Clustering Coefficient

The clustering coefficient expresses how tightly coupled neighbours of a given node to each other. The cluster coefficient of a given node i with degree k_i where L_i represents the number edges between the k_i neighbours can be formulated as [17]:

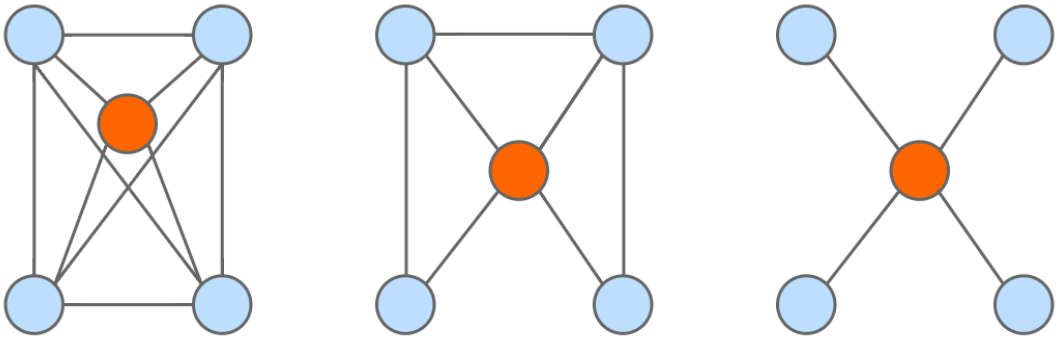
$$C_i = \frac{2L_i}{k_i(k_i - 1)}$$

Equation 3: Clustering Coefficient

Cluster coefficient is the probability that two neighbours of a given node link to each other hence the value is between 0 and1.

C_i = 0 if none of the neighbours have links between each other.

C_i = 1 if all the neighbours have links between each other and represent a complete graph.



C_i = 1 C_i = ½ C_i = 0

Figure 17: Cluster Coefficient of few graphs

Clustering coefficient can be used to measure the cohesiveness of a group of nodes [22].

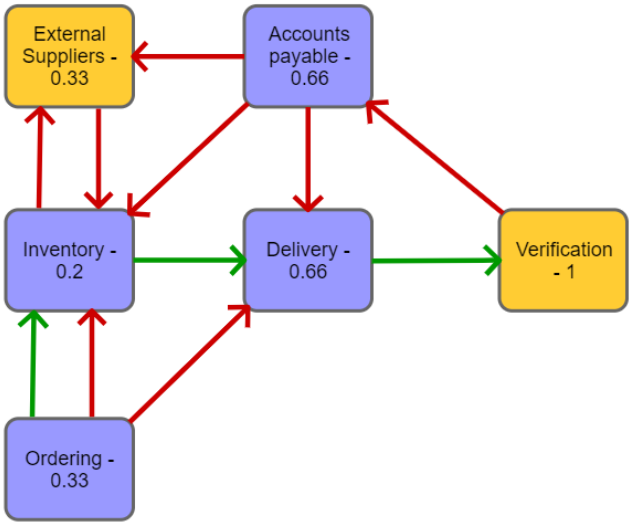


Figure 18: A microservice dependency graph with Clustering Coefficients of services

Figure 18 illustrates a dependency graph of a microservice system with corresponding clustering coefficient values. To calculate clustering coefficient for the Accounts payables service,

First, we need to find the Degree of the service. The service has three outgoing connections and one incoming connection. If we apply it to the Equation 2 which calculates Degree, where $k_i^{in} = 1$, $k_i^{out} = 3$ which gives the Degree of the service $k_i = 4$.

Then we need to find the number of connections between neighbouring services of the Account payable service. Accounts payable service has four neighbouring services, External suppliers, Inventory, Delivery and Verification services. Between External suppliers and Inventory services there are two connections, between Inventory and Delivery services there is only one connection and between Delivery and Verification services again only one connection. This adds up to 4 connections which is the number of connections between neighbouring services of the Account payables service.

Now we can apply the above calculated values to Equation 3 to find the Clustering Coefficient of the Account payable service.

$k_i = 4$

$L_i = 4$

$$C_i = \frac{2 * 4}{4(4-1)}$$

$C_i = 0.66$

Same way we can calculate Clustering Coefficient values of the other services as well.

2.1.6 Graph Databases

The emergence of new technology trends such as cloud computing, big data and other large scale web applications which generate massive amounts of data paves the path to a new database mechanism named as NoSQL [23]. The 'NoSQL' performs data storage and data retrieval mechanism completely different to a relational database [24]. NoSQL databases are adopted in heterogeneous fields and there are mainly four types of NoSQL databases. The four types are Graph databases, Key-value stores, Document stores and Column family stores. From these four NoSQL databases types, our focus will be on graph databases for this research.

The Graph database consider the concept of a graph. It stores and manages data using nodes and edges instead of storing them as relational data in a relational database [25]. A relational database has a very strict schema and it makes the process of adding new connections between objects difficult. The mechanism provides high efficiency when storing, manipulating and querying data in a distributed environment. Figure 19 represents an example graph database.

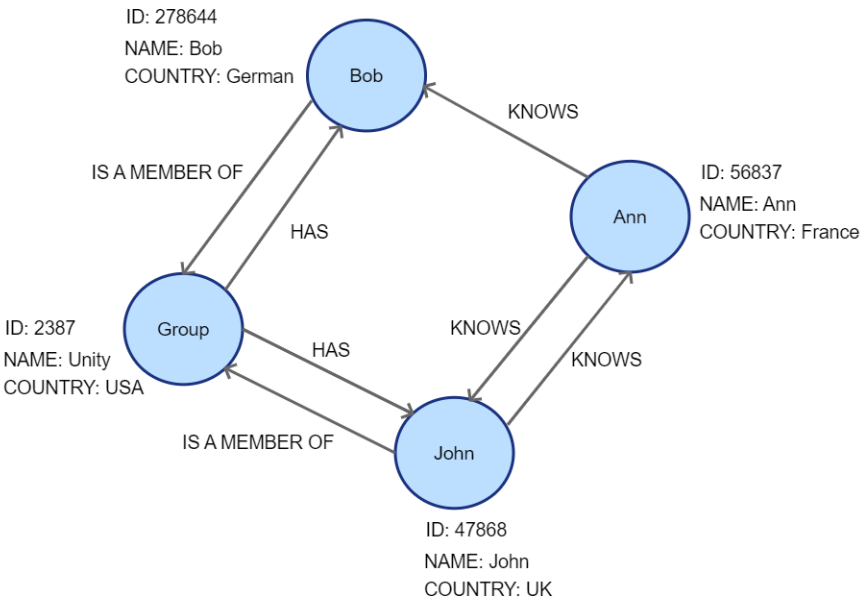


Figure 19: An example graph database

There are four nodes in the graph and each node caries properties Id, Name, Country. The nodes are interconnected by edges and there seven edges in the graph. In this database each edge carries contains properties such as Id, Label, From. Nodes have edges directed to both directions in the database. This is for the fact that, for an example, Bob is a member of group and thus the group has a member named Bob. Both these relationships are persisted in the database as directed edges to both directions.

A graph is a highly connected data structure hence graph databases are used to store data which are highly connected between them [26]. Famous social networking sites such as Facebook, LinkedIn use graph databases to store their users as nodes, relationship between users as edges and information of the users and relationships are stored as properties of nodes and edges respectively.

There are still limited number of studies have been carried out regarding the modelling of data in a graph database model hence there are no clearly defined rules provided yet in

modelling data in graph database domain. Some general rules based on the current studies are as follow [27]:

- Entities identified in the data can be represented using nodes of the graph
- Relationships between entities identified in the data can be represented by edges in the graph.
- Information of the entities are carried by the node properties.
- Information of the relationship between two entities are carried by the properties of the corresponding edge between two entities.
- Direction of an edge provides the semantic relationship direction between two entities.

Neo4j is an open source highly robust graph database, managed by Neo Technology [28]. It a transactional database which ensures ACID properties. The database is written using Java programming language. The main characteristics of the Neo4j database are intuitiveness, reliability durability, fast transactions, scalability and availability.

It is experimentally proved that, generally graph databases outperforms relational databases [29]. The retrieval time of graph databases are lesser than relational databases because, in order to find a data, relational databases go through all the data. When the data set becomes larger it increases the retrieval time as well to find matches. In order to find matches, graph database only go through the connected records and does not perform a full scan of all the records. So, when the data set become larger graph databases do not increases the retrieval time much as relational databases.

2.2 Related Work

2.2.1 Service Based Metrics

When service-oriented architecture started became popular and widely adapted in a distributed system manner, the design problems such as anti-patterns exist in such systems became a major concern and had to come up with metric based models to evaluate. In object-oriented programming (OOP), a major concern was how the core components such as classes, methods, modules, attributes should be designed, concepts such as inheritance should be incorporated.

In service-oriented architecture, another service layer was added on top of the above components which were in OOP programming. In service-oriented architecture, these services are the core components and it required metrics specifically designed for them to measure maintainability and performance concerns.

To address this, many researchers have been putting their efforts to adapt existing metrics and introducing new metrics models designed specifically for service-oriented systems. The anti-patterns are directly affecting the quality attributes such as maintainability and efficiency. In literature we can mostly see that the design implications are discussed in terms of quality attributes of service-based system.

Shim [38] proposed a hierarchical quality model which is presented as one such metrics collection for SOA systems. A hierarchical quality model designed for OO systems named as QMOOD (Quality model for Object Oriented design) has been extended to form the proposed model for SOA software. The model provides metric values corresponding to quality attributes of a SOA system such as effectiveness, understandability and reusability. To generate this quality attribute metrics values, in a lower layer of the hierarchy, it measures SOA design properties such as Coupling, Cohesion, Service granularity of the SOA system which has some implication regarding the patterns in the design but does not directly discusses the anti-patterns.

Senivongse [39] extends the same QMOOD model to create a quality model such that it mainly incorporates the quality attribute ‘maintainability’ which was not addresses in Shim’s quality model. Again, in this proposed method also system properties such as coupling and cohesion has been measured but does not directly imply regarding patterns in a SOA system.

Nik Daud [40] carried out a literature survey to discuss SOA structural attribute metrics which has been discussed on recent studies. The paper discusses structural attributes metrics such as coupling, cohesion and complexity with regard to SOA systems. Fifteen publications have been used for this review and the proposed metrics in them are then categorized as static and dynamic analysis. Further, they have classified them to a relevant stage in life cycle such as design phase or runtime and to a corresponding artifact such as WSDL or SOAP message. They have concluded the study mentioning that most structural attributes metrics are extracted using static analysis and there is a lack of studies on dynamic analysis used to measure structural attributes of SOA systems.

An approach which is more focused on evaluating a microservice system architecture to find the problems in the design has been proposed by Engel [41] called as MAAT framework. Their approach is tool supported and use runtime data of the system to generate the dependency graph of the system architecture. Then to evaluate identified principles from the literature such as small size of microservices, loose coupling and high cohesion, no cyclic dependencies and few other principles, they derived a set of automatically executable metrics using Goal Question Metric approach. Then the dependency model is evaluated using those metrics. The design hotspot can be identified using the visualizations provided by the tool. Even though this tool helps to find out design problems in a microservice system, the principle they identified are not discussed in correspond to anti-patterns. The source code of the tool is also not publicly shared.

Taibi [42] proposed a process mining technique to capture business functions from a monolithic application which then maps to suitable microservices based on similar behaviour but ensuring not to include cyclic dependencies and then set of metrics were

proposed to assess the quality of the decomposition. The main three metrics discussed were Coupling among services, how many classes exist in a microservice, how many classes have been duplicated among services. This approach provides some insights in to the quality of the architecture but not necessarily provide information regarding anti-patterns in the system.

2.2.2 Service Dependency Graph

For microservices systems it is very useful to visualize the service and their dependencies for debugging purposes, to get the latest architecture diagram, to find out hotspots in the architecture etc. For this, various tools have been introduced and usually they represent the microservice architecture as a dependency graph such that services are denoted by nodes and their communication dependencies as edges. Some methods generate the dependency graph dynamically using a distributed tracing tool.

One of the most prominent graph database technology widely adopted currently is Neo4j. It not only acts as a persistent storage but also provides visualization capabilities as well. These visualization capabilities of Neo4j have been widely adopted in areas such as visual analytics, social network analytics, fraud detection and many more. Neo4j provides tools and inbuilt data science library to evaluate graph models.

Akoglu [43] carried out a survey which aims providing anomaly and fraud detection using graph-based techniques as an overview. Also, they discussed how to analyse and explain the detected anomalies using graph theory concepts. In the survey they emphasized on four main reasons why graph-based techniques are beneficial in anomaly and fraud detection. First is, relationship and dependency among data, second is, ability to represent the dependencies using edges, third is, ability to extract information related to related components and finally, graph techniques can act as adversarial robust tools.

A microservice architecture can be analysed by extracting the relevant information from the systems. The methods of dynamic analysis have been studied in microservices area

such as the approach MicroART [32] which is capable of automatically extracting the deployment architecture of the system which is then semiautomatically refine it to the software architecture of the system using static and runtime information. Mayer [4] has presented a work which analyses a microservice based system by continuously extracting its information in three levels. The three levels are static, infrastructure and runtime information of the system:

- Each and every microservice has its own static information and it includes general description of the service, API details, domain model and organizational information because in a microservice system the architecture and the organization have a strong relationship.
- In a microservice architecture, services are deployed in their own servers or as containerized applications hence extracting the infrastructure level information is useful to analyse whether the resources are sufficient to address the infrastructure requirement of the service.
- The third category of architectural information includes communication relationships between services and this information can only be obtained at runtime. This extraction is long term and includes throughput, response time, error rate as well.

This information is used to visually present an overview of the system as graphs and charts. In dashboard one of the graphs shows the microservice system architecture with interactions between services. When a request is initialized to a service, the underlying information tracking system tracks all the services involved in that request and shows it as a directed graph.

One of the important use cases discusses in the paper is analysing the architecture to identify design problems with these visualized graphs. As an example, if graph shows an area with strong communications between services it indicates a high coupling area of the system which prevents scalability and independent deployment of the services.

The research is useful to identify how different levels of information related to a system can be utilized to present the different aspects of the system visually as graphs and charts as an overview. Even though one of the important use cases discussed in the paper is analysing the dependency graph to identify design problems in the system, the analysis of the dependency graph requires manual effort and the nature of the microservice complex architecture makes the analysis difficult. That is the reason why both statistical and visual information are useful in analysing a microservice system which is the main focus of our research work.

We can map a microservice system in to a graph database by an automated mean and apply graph algorithms on the graph database to identify poorly designed components. Ma [5] presents a tool which automatically generates a dependency graph of a microservice system with some automated analysis capabilities. The main three goals of the approach are visually presenting the dependencies between services, automatically detecting cyclic dependencies exist in the system and improving the service test coverage.

To generate the service dependency graph, first, all the necessary data such as service endpoints and service invocations are gathered using reflection mechanism implemented in services and then store them in a Neo4J graph database for further processing. Then the dependency graph is generated to show the links between services. The research proposes the concept of service invocation chain (SIC). SIC presents service invocation links between services for a specific request. The tool is capable of showing all the services involved in a SIC in the dependency graph when a user clicks on any node in the SIC. If a service test case failed for an invocation, the tool is capable of highlighting the SIC involved in the invocation so that the users can effectively find the root causes of the error.

After finding all the service invocation chains (SIC) in the system, the tool can detect cyclic dependencies in the system using *Tarjan's Strongly Connected Component algorithm* which is generally used to find strongly connected components of a directed graph [19].

The research shows how dependency graph and graph algorithms can be used together to generate valuable insights. But this research has only focused on one particular design problem, cyclic dependencies. There are many other anti-patterns that effect a system adversely that have not been covered in this research.

2.2.3 Anti-patterns in SOA

Palma [44] presented a study to quantifiably provide the impact of service patterns and anti-patterns on the maintenance and evolution of SOA systems using evolutionary historical data of FraSCAti. They considered two main effort metrics namely, 'number of changes' and 'size of changes' to measure the maintenance effort put to maintain and evolve a service by developers. One of the main observations of the study is, to maintain and evolve a service, more maintenance effort is required for the services with anti-patterns than the services which do not involve in anti-patterns.

Moha [37] proposed an approach for specification and detection of anti-patterns in service based systems named SOAD. They introduced a domain specific language to specify SOA anti-patterns and a mechanism which uses the specification to automatically generate anti-patterns detection algorithms. They applied the SODA with 10 SOA anti-patterns and on their experiment, they achieved a precision of 70% and a recall of 100%.

Taibi [12] conducted series of interviews with 72 developers and identified 265 bad practices experienced by them. Taibi analysed those bad practices and defined 11 microservice bad smells namely, API Versioning, Wrong service cuts, Cyclic dependencies, Inappropriate service intimacy, microservice greedy etc. The study emphasized that during monolithic to microservice transformation, wrong service cuts are the most adversely effecting bad practice for the maintenance of a microservice system later on.

3 METHODOLOGY

3.1 Introduction

Modern distributed services are large, complex, and increasingly built upon other similarly complex distributed services. Identifying design problems exist in these systems is not an easy task to perform. It involves the collection, interpretation, display and proper analysis of information concerning the inter relationships between services.

We can observe design problems in a distributed system by visually presenting the inter-relationships between services in a human readable way. In some cases, we might not have the architecture diagram for the system in hand or it might be out dated. Even we have the architecture diagram in our hand it is a difficult task to get insights in to the architecture by looking in to it if it is a significantly large system with many services.

Without any statistics it is hard to justify, do we need and why we need a refactoring of the system. We can trace the runtime information of the system and present interrelationships between services in a human readable way. But when the system grows in scale by adding new services such analysis cannot be done by just referring to a visualized dependency graph because the dependency graph will also tend to get untidy.

There are some proposed ways to analyse trace data and having insights in to the system to identify design problems. We have discussed some of them in section 2.2 –Related work. However, none of them make the analysis process convenient and do not provide any statistics to have quantifiable insights. To address this problem, there is a great need of a solutions which analyse properly traced and stored data in a visualized and a statistical way both.

3.2 Proposed Solution

We consider the challenges we discussed above in section 3.1 by referring to the literature and propose an approach to analyse a microservice architecture for anti-patterns with a tool support. The approach has four main steps (Figure 20). In the section 3.1 we discussed the potential problems of just depending on an architecture diagram or model in hand. So, we do not assume having the latest model of the architecture hence the first step of the proposed method is to obtain the relevant architectural data of the system.

Using these architectural data, we generate the architectural model which consists of the dependencies between services. Apart from the dependency model as a whole with all the services, we create models for individual traces. These models are then evaluated with the metrics we identified. The metrics are based on graph theory concepts and algorithms. As the last step, the dependency graph and the relevant charts with metrics are visualized so that developers can identify anti-patterns and take decisions regarding the architecture based on those visualizations and matric values.

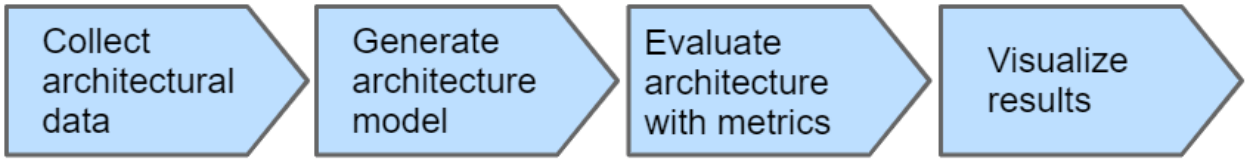


Figure 20: Microservice architecture evaluation process

In the following section 3.2.1, we discuss about the data extraction mechanism. In section 3.2.2, we discuss about analysing the selected anti-patterns with the corresponding matrices.

3.2.1 Data extraction

To generate the architecture model, we need to extract relevant information from the microservice system. This information includes, service names, end point URLs and relationships with other services. In this research we only focus on synchronous messages. Our intention is to completely avoid manual data extraction and make it automated so that users do not have to enter any information by themselves.

On the other hand, the performance of the productive system should not be affected or should be at a very minimal level in the data extraction process. To achieve these, we use a distributed tracing tool and collect the information at system runtime. Since the data extraction process is dynamic, we need to give a sufficient time span to gather all the communication data between services. There will be a minimal source code change for microservices which is a global configuration to integrate the distributed tracing tool. All these aspects are important for the practical usage of the proposed tool.

After allowing the tracing tool to trace the communications of the productive microservices system for sufficient time span, in the proposed tool, a module extracts these trace data from the tracing tool and store them in a graph database. When extracting the traces from the tracing tool, Trace Id, Span Id, Parent Span Id of the traces are considered. These data will then be stored in a graph database for further processing.

3.2.2 Evaluate dependency graph with metrices

We consider the five anti-patterns which we discussed in detail in subsection 2.1.2 based on existing literature for this research namely, The Knot, Nano service, Service chain, Bottleneck service and Cyclic dependency. To evaluate the architecture model, we derive metrics using graph theory concepts. These metrics are generated by performing graph algorithms provided by the underline graph database. For this research, three algorithms are used namely Degree centrality, Triangle counting and Strongly connected

components which the first one falls under Centrality algorithms and other two under Community detection algorithms [22].

In literature review we discussed how we can relate a microservice architecture model to a graph. We discussed some graph theory concepts in section 2.1.5 namely Degree, Clustering coefficient and Path distance and we derive metrics from them. We apply those metrics on the architecture model generated in step 2 to evaluate the model. Based on the results from evaluation, we can analyse and identify the anti-patterns exist in the model. Table 1 presents the anti-patterns, their corresponding metrics used to analyse and identify them and graph algorithms used to generate those metrics.

Identifier	**Anti-Pattern**	**Metric**	**Algorithm**
AP1	The Knot	M1: Clustering Coefficient	Triangle counting
AP2	Nano Service	M2: Degree Out	Degree centrality
AP3	Service Chain	M3: Path Length	(Custom)
AP4	Bottleneck Service	M4: Degree In	Degree centrality
AP5	Cyclic Dependency	(By visualizing)	Strongly connected components

Table 1: Anti-patterns and corresponding metrics

3.2.2.1 The Knot and Clustering Coefficient

In our architecture models there could be components which communicate among them in a very chatty manner. Those components can be considered as a Knot (AP1) as a whole which imply low cohesiveness and high coupling between services. To identify such kind of Knots, we use Clustering Coefficient (M1) of individual services.

As we discussed in the literature, Clustering Coefficient is the degree to which the neighbours of a node connected to each other. we can use clustering coefficient to evaluate the cohesiveness of a group of nodes connected [22]. If there are services with higher Clustering Coefficient, it implies that their neighbouring services are highly connected thus we can focus on such kind of service groups and find possible Knots.

Service	**Calculation**	**Clustering Coefficient**
External suppliers	$C_i = \frac{2 * 4}{4(4-1)}$	0.33
Account payables	$C_i = \frac{2 * 4}{4(4-1)}$	0.66
Inventory	$C_i = \frac{2 * 4}{4(4-1)}$	0.2
Delivery	$C_i = \frac{2 * 4}{4(4-1)}$	0.66
Verification	$C_i = \frac{2 * 4}{4(4-1)}$	1
Ordering	$C_i = \frac{2 * 4}{4(4-1)}$	0.33

Table 2: Clustering coefficients of a service dependency graph

Table 2 shows the Clustering Coefficients of all the services in Figure 22, service dependency graph. If we consider a threshold value as 0.3 for Clustering Coefficients, then we can notice that there five service which has Clustering Coefficient value more than that. Thus, we can say those services and their neighbours are less cohesive and tend to communicate in a very chatty manner resulting in tightly coupled system. We can focus on the relevant services and refactor them in order to make them high cohesive and loosely coupled.

3.2.2.2 Nano Services and Degree_Out

There can be services in traces which communicate with multiple services to fulfil a single request. The reason for that could be the low cohesion of related services which makes them Nano services (AP2). Since we consider individual traces when generating the architecture model, we can consider Degree_Out (M2) of each service in traces.

As we discussed in the literature, Degree of a node is the total of incoming and outgoing edges of the node. So, to find Nano services, if there are services in a trace with higher number of calls to other services, we can focus on those other services to find Nano services among them.

Service	**Degree Out**
Auth API	4
User API	4
Order API	2
Transactions API	0
Accounts API	0
Logger API	0

Table 3: Degree_Out values of a service dependency graph

Table 3 shows the Degree Out values of all the services in Figure 12, service dependency graph. Auth service and User service has significant Degree Out values compared to other services. Now we can narrow down finding the Nano services of the system by focusing only these two services. If we investigate the architecture further, we may find that the entry point of each trace is Auth service and it is normal to have many outgoing connections from it.

But when it comes to User service, we should check how the outing connections from it divided between different traces. That may give us the fact that, three invocations belong to the same trace and the other one to another trace. The three services, Transactions, Accounts and Logger might be behaving as Nano services in this scenario. Now we can concentrate on those three services to investigate the reason for low cohesion.

3.2.2.3 Service Chains and Path Length

To serve a single request, sometimes the request has to route through multiple services. This could be a result of low cohesion of related services and form a chain of services (AP3) for a single trace. We consider individual traces and evaluate their chain size based on the Path Length (M3) concept of graph theory. As we discussed in the literature, path length can be related to our model as the longest service call chain in a trace. Traces with higher path length values imply, longer service chains hence we can focus on them to make them high cohesive.

Trace identifier and colour representation	**Trace (longest)**	**Path Length**
T1 (brown)	Auth -> User-> Transaction/Accounts/Logger	2
T2 (orange)	Auth -> User -> Order -> Logger	3
T3 (green)	Auth -> Logger	1
T4 (blue)	Auth -> Order -> Logger	2

Table 4: Path Lengths of traces of a service dependency graph

Table 4 shows the Path lengths of traces in Figure 12, service dependency graph. For a trace we always consider the longest path of the trace to find Service chains. T3 trace may have significantly long service chain compared to other traces hence we can focus on Auth, User, Order and Logger services to make the relevant functionalities high cohesive thus reducing the length of the service chain and improve performance.

3.2.2.4 Bottleneck Services and Degree_In

In our architecture model, there could be services with high number of incoming service calls. The service calls could be from different traces. These services with high number of incoming calls are known as Bottleneck services (AP4) and considered as a design problem as we discussed in literature review.

Degree Centrality is an important concept when analysing a graph for influential nodes [22]. There the incoming and outgoing connections of nodes are looked at and assess how influential the nodes are based on their number of connections. Bottleneck services impact the design and the performance of a microservice system.

To evaluate the architecture model, we use the metric Degree_In (M4) of each service. A higher value for Degree_In of a service implies that it could be a Bottleneck service.

Service	**Degree In**
Auth API	0
User API	2
Order API	2
Transactions API	1
Accounts API	1
Logger API	4

Table 5: Degree In values of a service dependency graph

Table 5 shows the Degree In values of all the services in Figure 12, service dependency graph. Here we can ignore the fact that how the incoming connections are divided between different traces to a service. They could be from a single trace or from multiple traces which is irrelevant when finding bottleneck services. We can notice there are four incoming connections to Logger service which is a significantly higher value than of other services. This may imply that Logger service could be a bottleneck service. We can look in to the service and check whether any decoupling of functions is needed.

3.2.2.5 Cyclic Dependencies

If there are traces which form cyclic type patterns, they cause Cyclic Dependencies (AP5) in microservice system. Since we have individual traces in our architecture model, we can extract the service chains that form cycles and can visualize them.

4 IMPLEMENTATION OF MAIG

We implemented a tool that evaluates a microservice architecture. It considers the data collection requirement we discussed in subsection 3.2.1. Microservice Anti Patterns Insights Generator (MAIG) evaluates the architecture automatically and provides visualizations and corresponding metrics discussed in subsection 3.2.2 interactively. The tool is also capable of collecting trace data and generating the dependency graph of the system using the traced data.

The main goal of MAIG is to collect relevant runtime data dynamically, generate the latest dependency model of the microservice system with relevant metrics automatically to provide capability for the user to analyse and identify the anti-patterns exist in the system. The tool can be used in the development process to identify problematic areas in the microservice system in advance and also can be used with the productive system to get insights of the quality of the system. Figure 21 presents the architecture of the Microservice Anti Patterns Insights Generator.

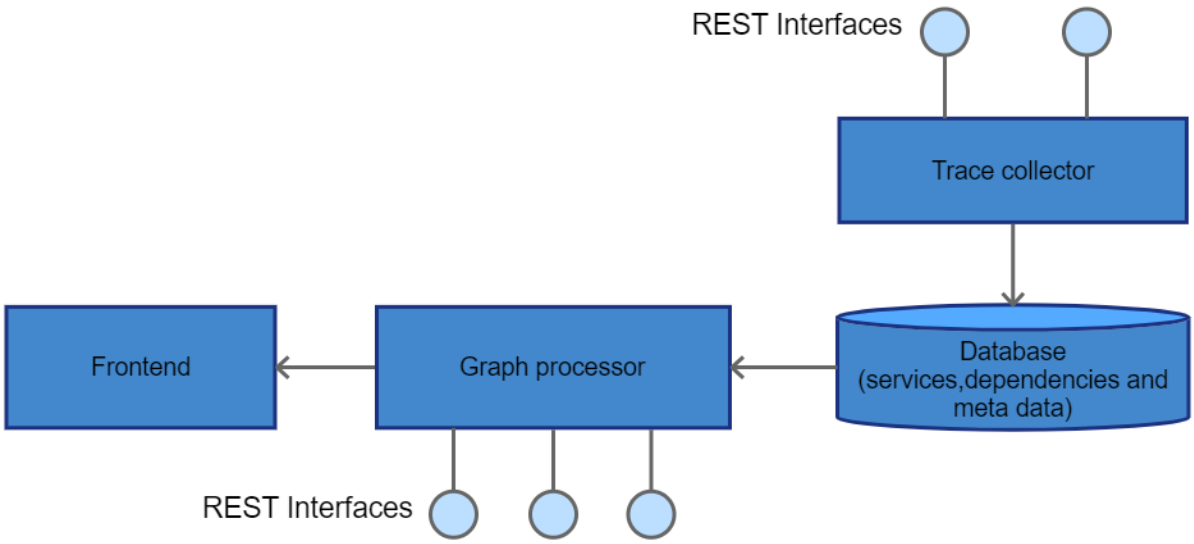


Figure 21: Microservice Anti Patterns Insights Generator (MAIG) architecture

4.1 Collect and persist data

Trace collector component provide an interface to collect trace data of the system. The trace data is captured at the runtime using Zipkin. Zipkin provides Open Tracing standard interfaces to provide different type of information related to traces. After allowing the Zipkin to trace the data in a given time span, we can invoke the *Trace collector interface* which in turn calls the Zipkin interface to retrieve trace data. The endpoint to retrieve trace data from Zipikin is *'/api/v2/traces'*.

Figure 22 shows a sample trace data provided by Zipkin interface. From that data, we consider below fields when creating the architecture model in database.

- id – a trace will have multiple spans and each will have a unique id
- traceId – multiple spans for single trace will have a unique traceid
- parentId – each span will have its parent span id
- kind – There are two possivle values, "CLIENT" and "SERVER". We only need spans with "SERVER" as that span is generated upon a service is serving a request to it
- localEndpoint: serviceName – Name of the service, serving a request to it
- tags: http.path – Full path of the endpoint consumed by a request (/api/service/endpoint)

Figure 22: A sample trace data provided by Zipkin interface

Trace collector component provides two interfaces to first get the trace data from Zipkin server and store them in a Json file and then another interface to insert the data in the Json file to Neo4j database. Table 6 shows the information regarding interfaces and their functionalities provided by the Trace collector component.

REST Endpoint	**HTTP method**	**Description**	**Response**
Api/TraceCollector/GetTraces	GET	Get trace data from Zipkin end-point and store them in a Json file.	OK
Api/TraceCollector/ InsertTraces	POST	Get trace data from Json file, create in-memory graph models and store them in Neo4j database.	OK

Table 6: Trace collector component interfaces

Upon retrieving the trace data, an in-memory object model is created to persist in a database as a graph model. Multiple object models are created. First one is the complete graph of all the traces. Other objects represent individual traces. This is for the convenience of data querying.

Within the time span which the system runtime data was collected, it is normal that system receives same request multiple times. We consider them as duplicate requests for our context because we only need trace data for unique requests. We don't perform any performance related evaluation and only need to identify the dependency graph as a whole and individual unique graph models related to unique request. So, we remove those duplicate traces when creating in-memory objects. If there are duplicate traces, we consider only the first trace with the corresponding trace id to insert in graph database.

After creating in-memory objects, they are inserted in to a Neo4j graph database using Cypher query language.

The algorithm 1, has a single parameter which accepts a list of objects for a single trace. The cypher unfolds the services and relationships from those objects and insert them in the graph database as a graph model. This will create services and their relationships in the database. This is for a single trace. For all the traces, the algorithm 2 loops the algorithm 1 and the list of in-memory objects we created above are passed corresponding to the trace.

Algorithm 1: Insert a single in-memory trace object in database

---

Data: List of service objects and their parent and child objects in a trace.

Result: Saved model in database.

1. Read each service object;
2. Check for the existence for the service in database;
    a. If exists: Ignore;

    b. If not exists: Create the service object in database;
3. Read parent service of the service;
4. Check for the existence of the parent service in database;
    a. If exists: Ignore;
    b. If not exists: Create the parent service object in database;
        i. Create the relationship of the service and parent service;
5. Check for the existence of the child services in database;
    a. Each child service: If exists: Ignore;
    b. Each child service: If not exists: Create the child service object in database;
        i. Create the relationship of the service and child service;

Algorithm 2: Insert all the in-memory trace objects in database

---

Data: All the traces with their list of service objects and parent and child objects.

Result: All the saved traces in database.

1. Read each in-memory trace object;
2. Invoke Algorithm 1: Pass in-memory trace object as a parameter

---

Now we have all the service information and their relationship information persisted in a Neo4j graph database.

4.2 Graph database

We use a Neo4j graph database to store the trace data. In our Neo4j graph database we have Services as nodes and communication links between them as edges. This makes it easier and efficient for us to query the database and get the data for dependency graph and metrices. Neo4j provides a Data science library to execute set of algorithms on the graph database. We are going to make use of them to generate some of the metrics we need in evaluation.

There could be multiple graph models in our database after inserting data from the Trace collector component. One model is for the dependency graph and all the other models present individual traces. We name them as *dependency graph model* and *trace models* respectively to easily distinguish between them. Instead of maintaining only the *dependency graph model*, we persist *trace models* at the same time because that approach is very convenient and efficient when querying the data for individual traces rather than maintaining individual trace data in same dependency graph model.

As an example, when we need to present an individual trace visually, we only need to pass the Trace Id for it and directly fetch the corresponding trace. Table 7 presents the properties of nodes and edges in both type of graphs, existence of values for them and which properties of Open tracing standard interface they are extracted from.

Property	**Node/Edge**	**Has Value**		**Reference property of Open tracing standard interface**
		Dependency graph model	**Trace models**	
id	Node	✓	✓	None (Assigned by Neo4j)
name	Node	✓	✓	localEndpoint: serviceName

traceid	Edge	×	✓	traceId
endpoint	Edge	×	✓	tags: http.path

Table 7: Properties of nodes and edges of the dependency graph

4.3 Process data

We use persisted data in Neo4j graph database to generate dependency graph and other metrics. *Graph processor* component is capable of retrieving relevant data by issuing cypher queries to graph database. The component provides RESTful interfaces to fetch data to create dependency graph, to generate metric values degree, clustering coefficient, path length. Table 8 presents the RESTful interfaces of *Graph processor* component.

REST Endpoint	**HTTP method**	**Description**
api/data/dependencygraph	GET	Get data to display dependency graph
api/data/degrees	GET	Get degree_in, degree_out of nodes for degree metrics
api/data/clusteringcoefficients	GET	Get clustering coefficients of nodes for degree metric
api/data/pathlengths	GET	Get path length of individual traces for path length metric
Cyclic dependencies	GET	Get data to display cyclic dependencies

Table 8: RESTful interfaces of Graph processor

We will now discuss the queries used to generate dependency graph and other metrics. All these queries are executed against the graph database as cypher queries.

Query to get data for dependency graph

Query 1: Get data for dependency graph (in cypher)

Input data: None.

Result: Services and their relationship details.

1. MATCH (node1 {traceid:'null'})-[relationship]-(node2 {traceid:'null'});
2. RETURN node1,node2, relationship;

Query to get Degree_Out values of services

Query 2: Get Degree Out values of service nodes (in cypher)

Input data: None.

Result: Services and their corresponding Degree Out values.

3. MATCH (s:Service);
4. WHERE s.traceid = 'null';
5. WITH s AS service;
6. SIZE((s)-[]->()) AS degree_out;

7. RETURN service, degree_out;

This selects the services where ‘traceid’ is of value ‘null’ which eliminates individual trace models in the graph and only consider the dependency graph model. Then the Degree Centrality graph algorithm from Neo4j Data science library is used to get the degree_out of service nodes.

Query to get Degree_In values of services

Query 3: Get Degree In values of service nodes (in cypher)

Input data: None.

Result: Services and their corresponding Degree In values.

1. MATCH (s:Service);
2. WHERE s.traceid = 'null';
3. WITH s AS service;
4. SIZE((s)<-[]-()) AS degree_in;
5. RETURN service, degree_in;

This selects the services where ‘traceid’ is of value ‘null’ which eliminates individual trace models in the graph and only consider the dependency graph model. Then the Degree Centrality graph algorithm from Neo4j Data science library is used to get the degree_in of service nodes.

Query to get Clustering Coefficient values of services

Query 4: Get Clustering Coefficient values of service nodes (in cypher)

Input data: None.

Result: Services, their corresponding trace ids and clustering coefficient values.

1. CALL gds.localClusteringCoefficient.stream('graph');
2. YIELD nodeId, localClusteringCoefficient;
3. RETURN gds.util.asNode(nodeId).traceid as traceid, gds.util.asNode(nodeId).name AS name, localClusteringCoefficient as clustering_coefficient;

Query to get Path length values of services

Query 5: Get Path Length values of service nodes (in cypher)

Input data: None.

Result: All the traces with their trace id and longest path length.

1. match p=(par:Service)-[r:calls*1..10]->(ch:Service);
2. where par.traceid<>'null' and ch.traceid<>'null';
3. return DISTINCT par.traceid as traceid, max(length(p)) as length;

Query to get data for Cyclic Dependencies

Query 6: Get data for Cyclic Dependencies (in cypher)

Data: Input data.

Result: Services and their relationships that form cyclic dependencies for each trace.

1. MATCH (e);
2. WHERE e.traceid<>'null' and SIZE ((e)<-[:calls]-()) <> 0
   AND (()<-[:calls]-(e)) <> 0;
3. MATCH path = (e)<-[:calls*]-(e)
4. RETURN e, path;

4.4 Frontend application

The MAIG frontend application has been designed as a single page application using Angular framework. The front-end application invokes *data processor* component via REST calls. The application has an interactive dashboard with a *Reference* section and other sections for visualizations and statistics. The *Reference* section is loaded first with all the information on how to use the system and how to analyse relevant graphs and statistics to identify anti-patterns. This makes the application very intuitive to the user. Figure 23 shows the dashboard of the application.

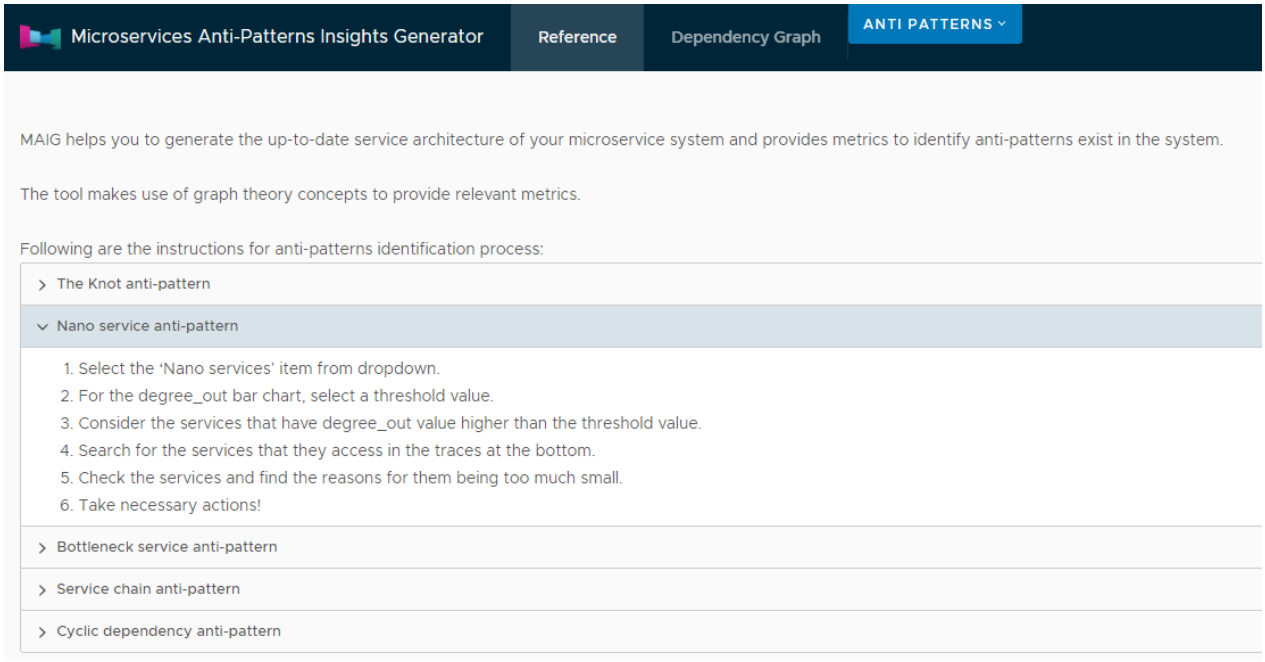


Figure 23: Dashboard of the MAIG

4.5 Data visualization and metrics

The service architecture of a microservice system is visualized as a dependency graph. Nodes in the dependency graph represent services and edges represent communication links. The nodes are labelled with corresponding service names. The graph is generated as a directed graph hence communication directions between services are presented with arrows on edges. Figure 24 shows an example of a service dependency graph of a microservice system.

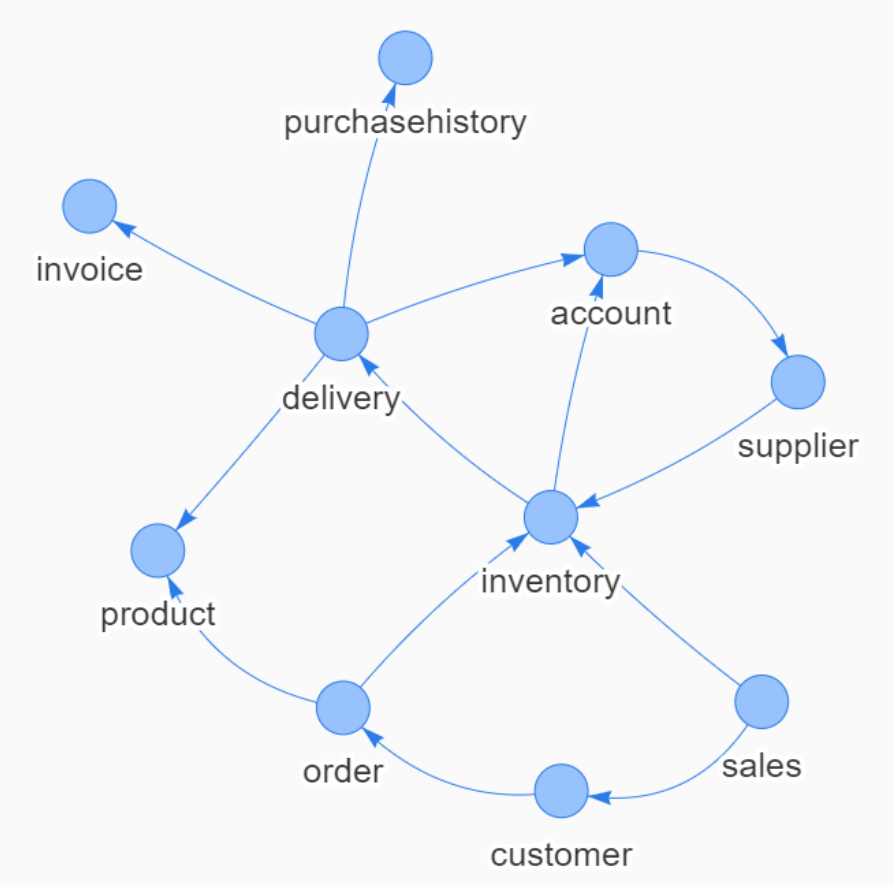


Figure 24: An example service dependency graph generated by MAIG

The *Anti Patterns* dropdown list on the top navigation bar list the anti-patterns to be analysed. When we access the item *'The knot'*, it brings up the page with the bar chart for Clustering Coefficients of the services. At the bottom, service dependency graph is displayed to make the analysis convenient besides the bar chart. Figure 25 shows an example screen of the *'The knot'* section where we analyse the architecture to find Knots.

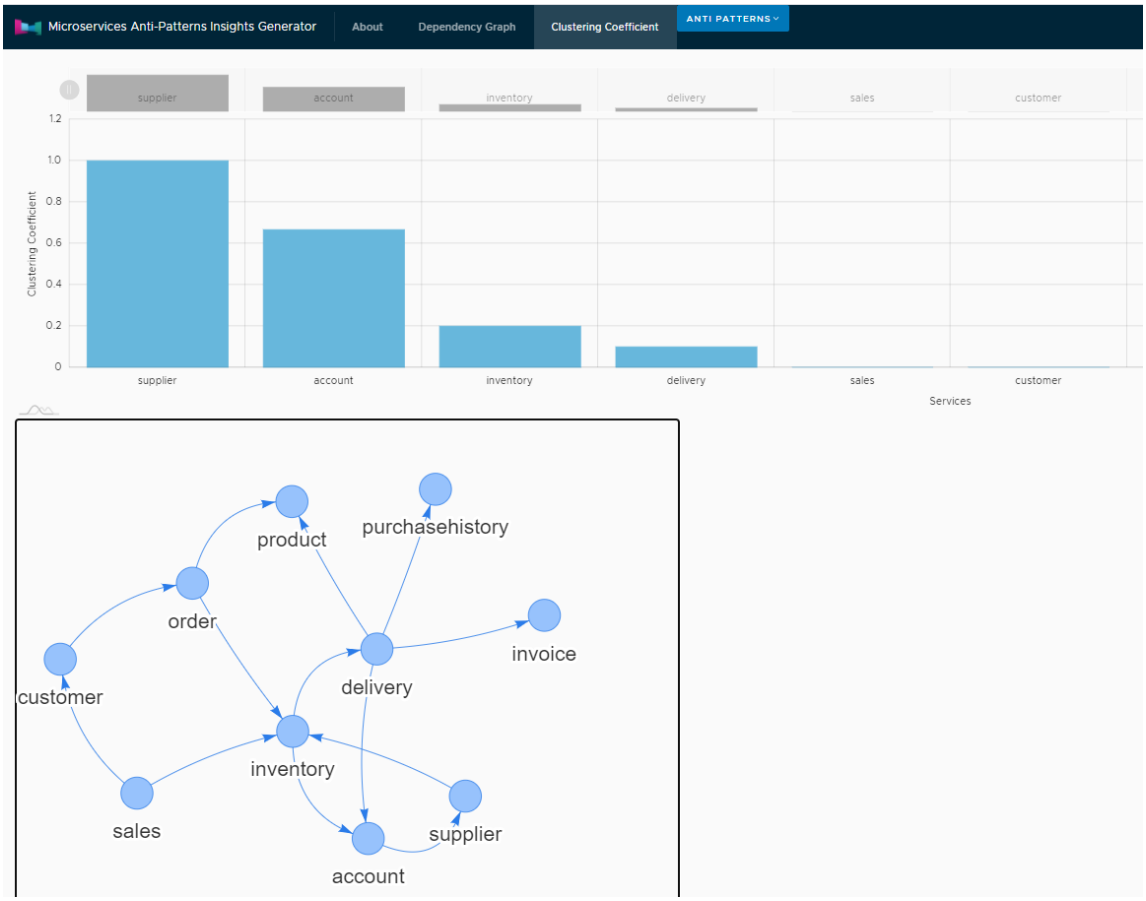


Figure 25: MAIG section to identify Knots in the service architecture

The other items in the dropdown bring up the corresponding bar charts and graph components same as the previously discussed *'The Knot'* section. In the *'Nano services'* section when we click on a bar of a service, it fetches and visualize the corresponding

individual outgoing traces from the service separately. In the *'Bottleneck services'* section the same process will fetch and visualize all the incoming traces to the service. This is displayed as a single graph. *'Service chains'* section also follows the same process and fetch individual traces and visualize them as graphs in addition to the bar chart. *'Cyclic dependencies'* section does not provide any statistics but visualizations of cycles.

4.6 Zipkin configuration

We use Zipkin as the distributed tracing system to trace the request data at runtime. There are two options to start an instance of an Zipkin:

Docker:

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

Java

```
curl -sSL https://zipkin.io/quickstart.sh | bash -s
java -jar zipkin.jar
```

Now we need to configure Zipkin in each service project. Since we use *.NET Core framework* to develop services, in each service project, we first install *zipkin4net* library from *NuGet Gallery*. Then we do the below configuration in *Startup.cs* file in the service project. This is a global configuration and the only change we need to configure Zipkin in a service project.

Configure method:

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
        var applicationName = "<ServiceName>";

        var lifetime = app.ApplicationServices.GetService<IHostApplicationLifetime>();
        lifetime.ApplicationStarted.Register(() => {
                TraceManager.SamplingRate = 1.0f;
                var logger = new TracingLogger(loggerFactory, "zipkin4net");
                var httpSender = new HttpZipkinSender("http://<host>:9411", "application/json");
                var tracer = new ZipkinTracer(httpSender, new JSONSpanSerializer());
                TraceManager.RegisterTracer(tracer);
                TraceManager.Start(logger);
        });
        lifetime.ApplicationStopped.Register(() => TraceManager.Stop());
        app.UseTracing(applicationName);
}
```

ConfigureServices method:

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddHttpClient("Tracer").AddHttpMessageHandler(provider =>
                TracingHandler.WithoutInnerHandler("<ServiceName>"));
}
```

5 EVALUATION AND DISCUSSION

In this section we focus on evaluating a microservice system with the tool we proposed to identify the anti-patterns in the system. The microservice system is a sample application we developed specifically for this evaluation that consists of 10 services and 1 front-end application and we have deliberately introduced five anti-patterns to the system. The tool is integrated with this microservice system for evaluation purposes.

The objective of the evaluation is to present the capability and the automated nature of proposed tool MAIG in identifying anti-patterns accurately using the service dependency graph and the graph theory metrics generated by the tool. Here we discuss in detail how to analyse the results produced by tool to identify anti-patterns.

5.1 Capturing trace data of the system under test (SUT)

First, we run the Zipkin server and then we perform various requests to the SUT for some time. The time period has no limitation and Zipkin traces all the requests performed in that time period. Since we need to capture all the system dependencies, it's required to allow system under test to function for a sufficient time period such that the Zipkin server will trace all the requests that can be performed for the system. As discussed earlier we use Swagger to perform the requests.

After allowing the SUT to function for a sufficient time, we invoke the end-point *'Api/TraceCollector/GetTraces'* from the *Trace collector* component to store them in a Json file. The response should be 'OK' if successful. We then invoke the end-point *'Api/TraceCollector/InsertTraces'* to create the in-memory graph models and store them in Neo4j database.

Now we have persisted all the dependency information in the graph database to generate relevant dependency graphs and metrics for evaluation.

5.2 Evaluation

5.2.1 Dependency graph evaluation

One of the main functionalities of MAIG is to generate the dependency graph of the microservices system. To visualize the dependency graph for the SUT, go to the 'Dependency Graph' tab of MAIG. Figure 26 shows the dependency graph for the SUT.

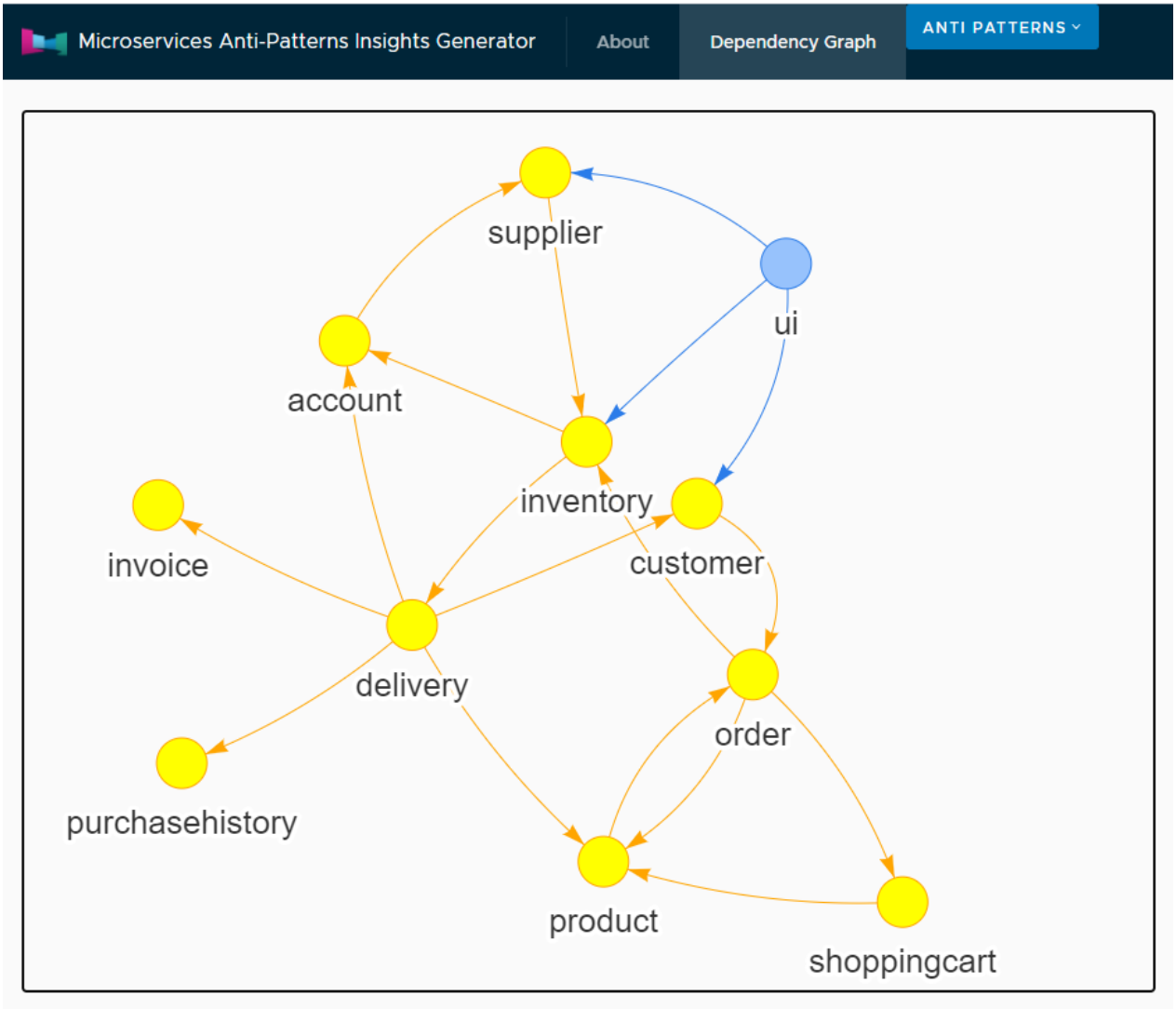


Figure 26: Dependency graph of the SUT

The dependency graph shows 10 services with a frontend application (UI) and their inter-dependencies. Frontend application and the services are coloured differently to identify them intuitively. As discussed earlier, it is required to allow Zipkin to trace the communications for a sufficient time period such that it traces all the dependencies between services. If that is the case, now we have the complete dependency graph of the microservice system.

We can have insights of the microservice architecture to some extent by just observing the dependency graph. The dependency graph shows a directed link between two services if the caller service invokes at least one end point of the invoked service. The idea is to provide user with simplified relationship details.

The *delivery service* is dependent on five service as it has four dependency links to external services. A single link may imply one or more requests to the other service which cannot be clarified at this stage. Also, it cannot be clarified yet, whether all five links are for a single trace or for multiple traces. If the majority of the links are for a single trace then there is a possibility of external services being *Nano services*.

The *inventory service* and the *product service* both has three incoming dependencies from external services which implies that there is a possibility of *inventory* and *product services* being bottleneck services. *Supplier, account, inventory, delivery, customer, product, order, shoppingcart services* tend to be in a chatty behaviour. This implies an existence of possible *Knots* in the system.

In this way we can use the dependency graph to have insights in to the architecture and find possible design problems to some extent. If we are asked to provide the up-to-date service architecture of the system, then this graph can be presented. To obtain detailed insights of architecture with metrics, we can utilize the other functionalities provided by MAIG. For a small graph with few services, it is possible to perform the above analysis by referring to the dependency graph. If the microservice system has too many services, that analysis would be hard and we need to check metrics with relevant visualizations to gain clear insights.

5.2.2 Metrics evaluation

The navbar of the application has a dropdown with all the anti-patterns as items. When an item is selected, the user will be redirected to an interface with the bar chart of a relevant metric. Each interface provides relevant visualizations at bottom. *Cyclic dependency* interface only provides a visualization without any metric bar chart.

We can consider a threshold value for each metric and consider the services which have higher metric values than the threshold value for analysis. This threshold value supposed to be depended on the size of the application and up to the user to decide. Larger the application, larger the threshold value considered.

5.2.2.1 Clustering coefficient metric to identify Knots

We select the dropdown item *'The knot'* to analyse and identify chatty services in the system. Figure 27 show the bar chart of services with their corresponding *clustering coefficient* values and at the bottom the dependency graph of the system to make the analysis process convenient.

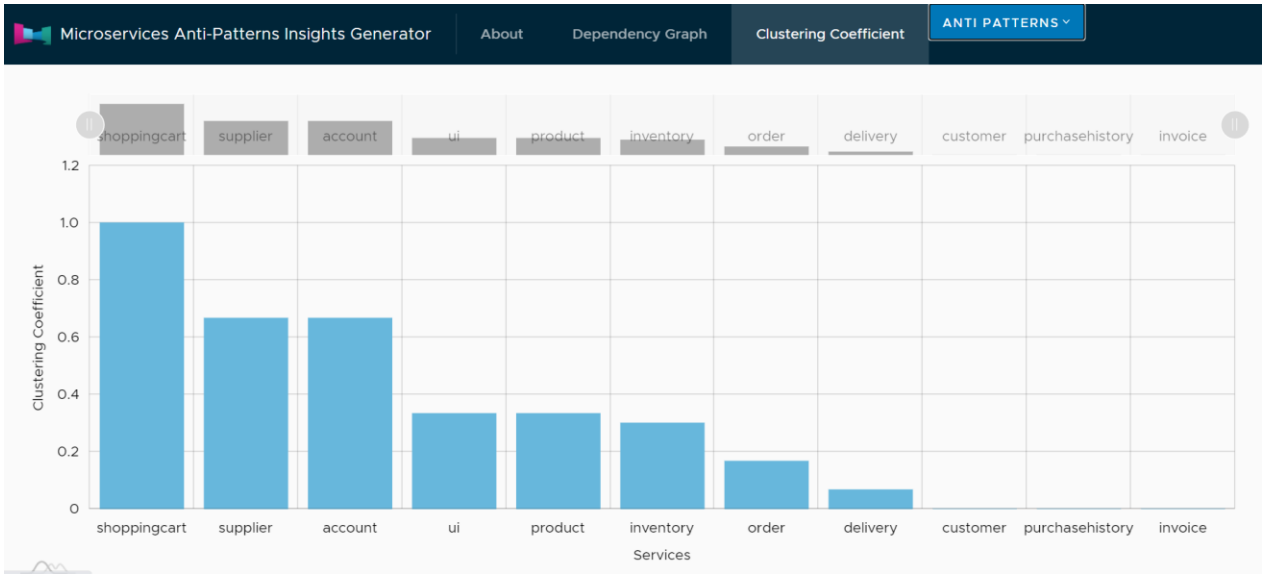


Figure 27: Clustering coefficient bar chart for the SUT

Based on the bar chart, there are three services that have value 0 for clustering coefficient. *Customer, Invoice and PurchaseHistory* has 0 for clustering coefficient which indicates the neighbouring services of those services do not interact with each other. The relationship links might represent a single trace or multiple traces. The directions of the relationships are not considered here. Only the relationships are considered.

The *clustering coefficient* of the above services being 0 implies that the services involved are not in a chatty behaviour which in return implies that the involved services are not contributing to a *Knot*. The *Customer service* interacts with *Order service* and *Order service* do not interact with any other services that interact with *Customer service* hence the *clustering coefficient* of *Sales service* is 0 and all the services involved do not contribute to a *Knot*.

If we consider the threshold value as 0.3 for *clustering coefficient*, then there are five services that have the associated clustering coefficient values larger than or equal to threshold value and frontend application also exceeds the threshold value. These services and their neighbouring services tend to form a *Knot* in the system.

If we look in to the *Inventory service*, it interacts with four services and with the frontend application. The neighbouring services are *Supplier, Account, Order and Delivery*. *Account service* interacts with *Supplier service* and *Delivery service* interacts with *Account service*. We need to investigate why these neighbouring services are in a chatty behaviour and consider actions to make them loose coupling and high cohesive which in return minimize the chattiness.

5.2.2.2 Degree_Out metric to identify Nano services

The *'Nano services'* item of the dropdown in the navbar brings up an interface with Degree_Out metric values of services as a bar chart. This is used to indicate the user whether there is a possibility of existence of too much small services such that a service (secondary service) cannot provide a complete response to the caller service (primary

service) because the business logic is not well bounded context which in return makes caller service to invoke some other services to complete the response. This makes invoked service a *nano service* thus should consider a redesigning.

The user is required to investigate relevant services because an invoked service could have a well bounded context hence not a *nano service* and can be ignored. Figure 28 presents the Degree_Out bar chart of services of the SUT and at the bottom it displays corresponding out going relationships of each service when user clicks on the corresponding bar in the chart for the service.

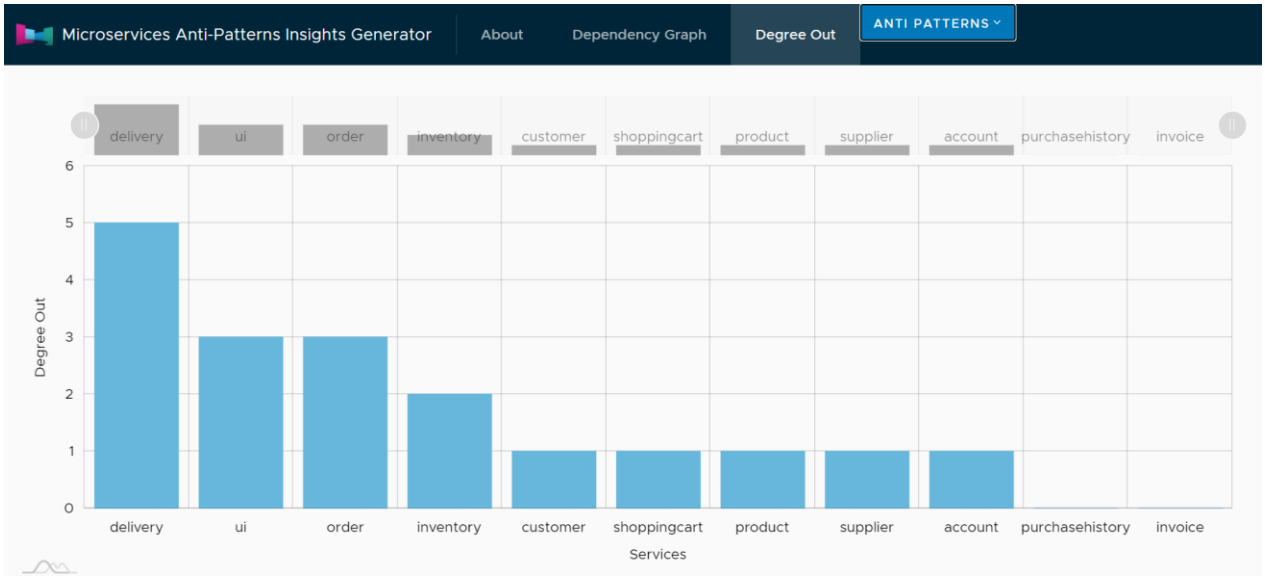
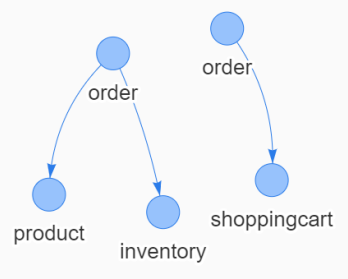


Figure 28: Degree_Out metric bar chart for the SUT

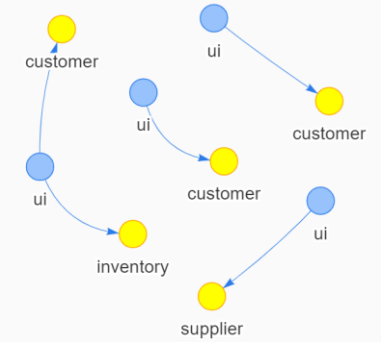
Invoice and *PurchaseHistory services* each have *Degree_Out* value 0. This indicates that those services have no outgoing connections to any other services thus can be ignored. If we consider a threshold value as 3 for *Degree_Out*, then there are two services which have *Degree_Out* values equal or larger than that. *Delivery* and *Order services* have *Degree_Out* values 5 and 3 respectively. Frontend application also has a *Degree_Out*

value 3. We can further investigate how the relevant services are connected using visualization capabilities provided. For that, in the chart, click on the bar of the particular service or frontend application. Figure 29 show the visualizations for outgoing connections from *Delivery*, *Order* and Frontend application to other services trace wise.

Visualization for outgoing connection from Delivery service



Visualization for outgoing connection from Order service



Visualization for outgoing connection from Frontend application

Figure 29: Visualization for outgoing connections of SUT

It can be noticed from the visualizations that the *Order service* has two outgoing traces. The *Degree_ Out* value 3 of *Order service* is distributed among two traces hence we can ignore investigating invoked endpoints of services *Product, Inventory* and *Shoppingcart* for this scenario. The *Degree_Out* value of Frontend application also distributed among four traces hence we can ignore the corresponding invoked services there as well.

The *delivery service* has 5 outgoing connections to external services and visualization shows they all belong to a single trace which implies that to form the complete response for the *Order service* request, the individual responses from the external services *Account, Invoice, Customer, Product* and *PurchaseHistory* are required. This indicates that there could be a possibility of existence of *Nano services* among the invoked services hence an investigation needs to be carried out for each service end point.

5.2.2.3 Degree_In metric to identify Bottleneck services

The *'Bottleneck services'* item in the dropdown redirects user to an interface with bar chart for *Degree_In* metric of services. This bar chart provides how many incoming connections each service has from other services. If the incoming connections are significantly high for a service, it indicates that the service acts as a bottleneck service in the system hence the service needs to be investigated further and decouple functions to multiple services if required.

Figure 30 presents the *Degree_In* bar chart of services of the SUT and at the bottom it displays corresponding incoming relationships of each service when user clicks on the corresponding bar in the chart for the service.

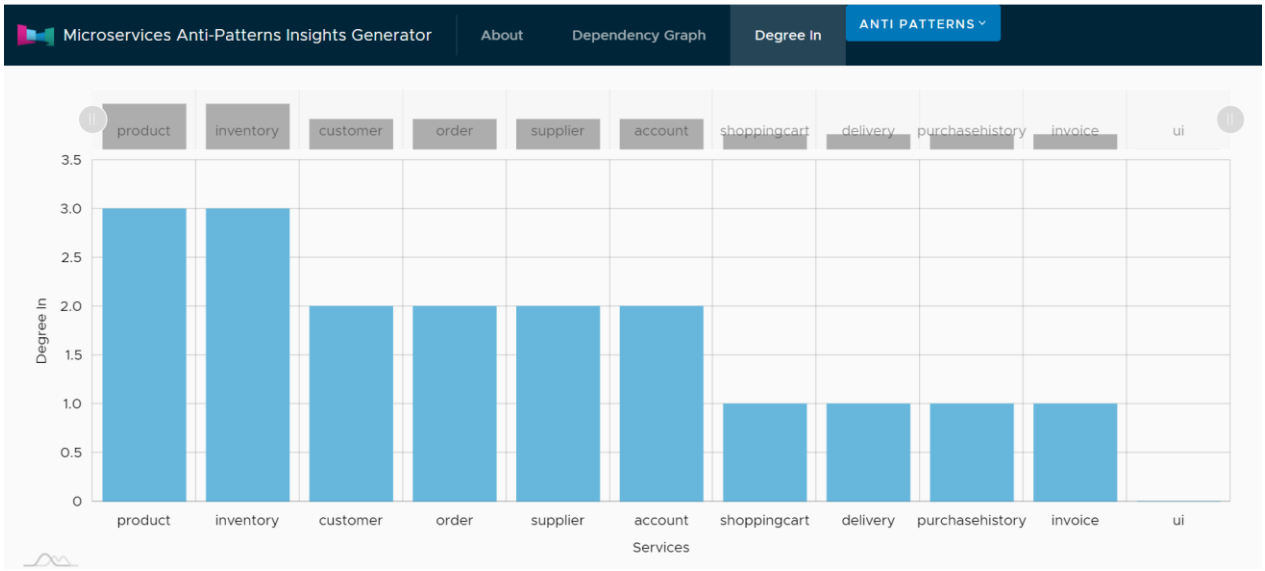
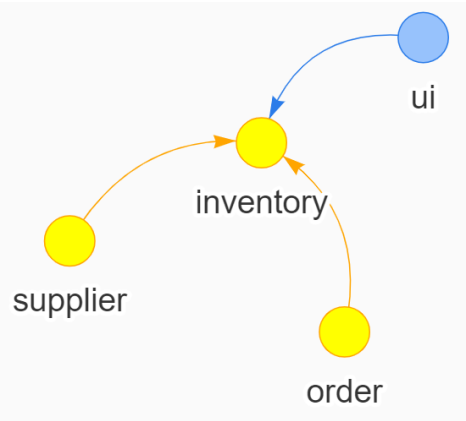


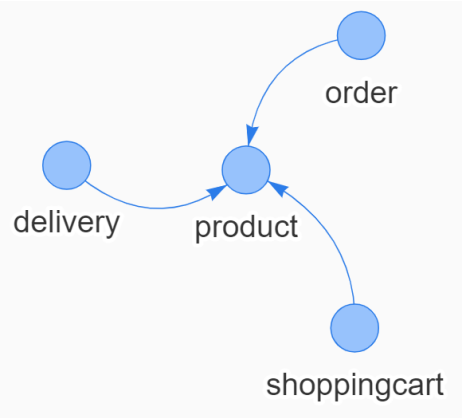
Figure 30: Degree_In metric bar chart for the SUT

If we consider a threshold value as 3, based on the bar chart there are two services which has *Degree_In* metric value of 3 which needs to be further investigated for *bottleneck services*. No other service can be found which has a higher Degree In value than the threshold value. All the other services that have lower *Degree_In* values than the threshold, are ignored from further investigation.

Inventory and *Product services* both have a *Degree_In* value of 3. To further investigate from which services they get incoming connections, we can use the visualization capability. Click on the corresponding bar in the chart of a service to get the visualization of incoming connections. Figure 31 shows the visualization for incoming connections to *Inventory* and *Product services* from other services.



Visualization for incoming connections to Inventory service



Visualization for incoming connections to Product service

Figure 31: Visualization for incoming connections to Inventory service of SUT

Inventory service has incoming connections from *Supplier service*, *Order service* and from Frontend application and for *Product service* from *Order*, *ShoppingCart* and *Delivery services*. We need to investigate why all these external services and frontend applications have to access or go through *Inventory* and *Product services* this way by investigating the two services individually. A reason could be tightly coupled business logics or functions which are supposed to be in sperate services. This might result in a

huge incoming traffic to these two services which might in return result in a performance hit hence needs to consider a decoupling of business logics to separate services if required.

5.2.2.4 Path length metric to identify Service chains

The interface for the *Path length* metric bar chart of the services can be brought up by clicking on the *'Service chains'* item in the dropdown. This chart considers all the unique requests that can be made to the system. The X axis denotes the Trace Ids of the first occurrence of each unique request and the Y axis denotes the length of the longest path of a particular request through services. These paths form chains of services and longer service chains could result in Service chain anti-pattern. Figure 34 presents the *Path length* bar chart for the SUT.

The chart indicates that there are 4 unique requests that can be made to the system. If we consider the threshold value as 4, then there are two unique requests *Trace 1* and *Trace 2*, which has the *path length* values equal or more than the threshold and two unique requests *Trace 3* and *Trace 4*, less than the threshold. We can ignore the latter two from further investigation. To investigate which services are involved in *Trace 1* and *Trace 2*, we can use visualization capabilities provided. For that, click on the corresponding bar in the chart for a trace and it will visualize the complete trace at the bottom of the interface.

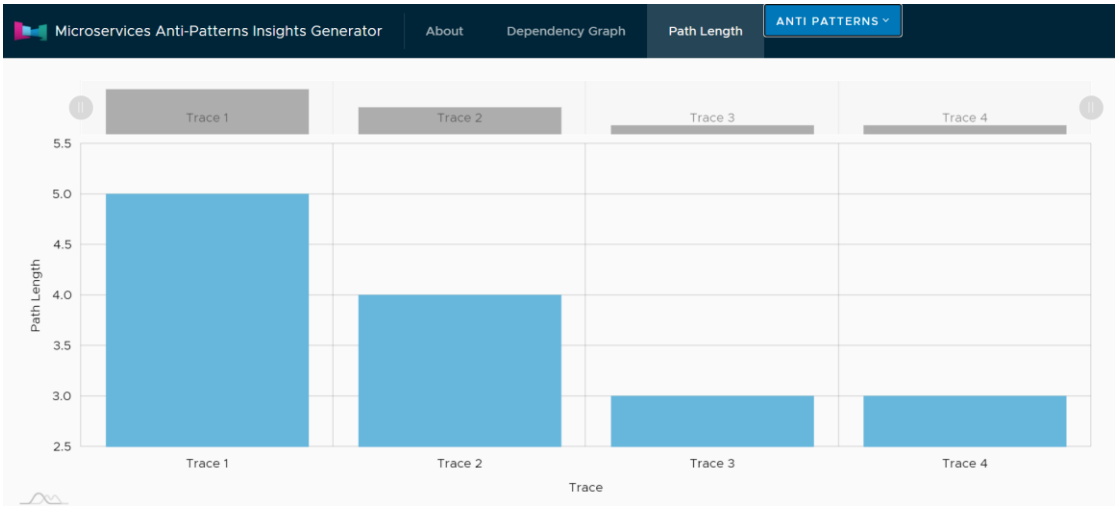
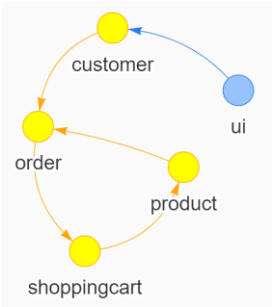
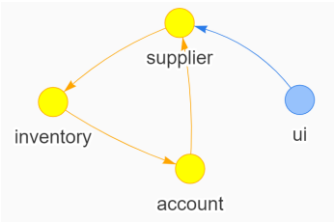


Figure 32: Path Length metric bar chart for the SUT

Figure 33 presents the visualizations of individual traces for Trace 1 and Trace 2.



Trace 1



Trace 2

Figure 33: Visualization for unique traces of SUT

For *Trace 1*, the request access four services in a chained manner to get the response. The request path is as follows:

UI -> Customer -> Order -> ShoppingCart -> Product -> Order

This makes five connections as a chain between services and this chain itself covers the complete trace. Now we need to find out which endpoints of the services are invoked in the chain. For that, hover the cursor over the incoming link of a particular service. A tooltip is displayed with the information of TraceId and Endpoint invoked. The invoked endpoints of the services are as follows:

Customer/endpoint_3, Order/endpoint_2, ShoppingCart/endpoint_1, Product/endpoint_3 and finally Order/endpoint_3.

We should investigate these service endpoints to figure out why the request has to go through all these endpoints to get a response. A reason could be, the services are less cohesive so that the business logic is distributed among the services. If that is the case, we must consider making the services high cohesive by encapsulating relevant business logic in lesser number of services so that the request path length becomes lesser than the threshold. The same investigation needs to be carried out for *Trace* 2 as well.

5.2.2.5 Cyclic dependencies

The cyclic dependencies exist in the system can be visualized by accessing the *'Cyclic dependencies'* section using dropdown. Figure 34 presents the visualization of *cyclic dependencies* of the system.

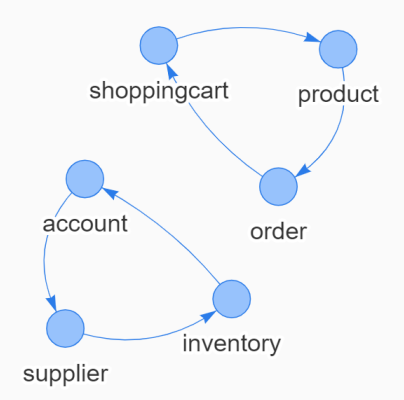


Figure 34: Visualization of cyclic dependencies of SUT

Based on the visualization we can notice that there are two *cyclic dependencies* exist in the system. These two cycles belong to two separate traces. To get rid of this design problem we need to investigate the relevant endpoints of the services. If we consider *Order, ShoppingCart, Product, Order cyclic dependency*, the endpoints involved are *ShoppingCart/endpoint_1, Product/endpoint_3, Order/endpoint_3 respectively*. We should refactor the relevant business logic such that the cycle will not exist anymore. We should perform the same for *Supplier, Inventory, Account, Supplier* cycle as well.

6 CONCLUSIONS

6.1 Contributions

In this research we proposed a tool-based approach to analyse and identify anti-patterns exist in a microservice system. The approach is purely based on graph theory. The distributed nature of a microservice system and dependencies among the services allows us to perceive a microservice architecture as a graph. The services in the microservice system can be perceived as nodes in a graph and dependencies among the services as edges.

Based on this if we map a microservice system to the corresponding dependency graph we can apply various graph concepts and algorithms to analyse that dependency graph such as *Degree, Clustering coefficient and Path distances*. In the thesis we showed how some of the service based anti-pattern can be directly related with these graph concepts.

As an example, we showed that the anti-pattern *Bottleneck service* can be related with the graph theory concept *Degree*. In *bottleneck service* anti-pattern, many services access a particular service and because of the massive traffic to the service it acts as a *bottleneck service*. *Degree* in graph theory refers to how many links a node has with other nodes. From this we defined a property called *Degree_In* which only considers the incoming directed links to a node and used that as a metric value calculated by performing a degree centrality algorithm on the dependency graph of the microservice system. The value calculated for the metric *Degree_In*, gives how many incoming dependencies a particular service in a microservices has an that gives an insight on whether the service tends to be a *bottleneck service*.

Such that we evaluated and identified *The Knot* anti-pattern with *Clustering coefficient* as the corresponding metric, *Nano service* anti-patterns with *Degree_Out* metric and *Service chain* anti-patterns with *Path distance* metric. *Cyclic dependencies* are visualized

without any metric. But higher *Clustering Coefficient* metric value of a service hints the possibility of the service being in a cycle as well.

We developed a tool called *Microservice Anti-patterns Insight Generator (MAIG)* which is capable of generating the up-to-date service dependency graph of the microservices system using runtime data of the system. The tool generates relevant metrics and visualizations so that the user can analyse them to get insights in to the system and thereby identify anti-patterns. Metric values are presented as bar charts and corresponding traces are visually presented. In this research we mainly focused on five anti-patterns, namely: *The knot, Nano service, Bottleneck service, Service chain and Cyclic dependency* anti-patterns.

The metrics are based on graph theory concepts. The graph theory concepts used in the research are *Degree, Clustering coefficient and Path length*. The system traces are mapped as graph models such that a node in a graph represents a service in the system and a directed edge in a graph represents the relationship between two services. We used *Zipkin* as the distributed tracing tool to capture the traces at runtime. As the database we use *Neo4j* graph database in order to utilize their data science library to efficiently run queries on the database.

In this research Neo4j graph database played a vital role. We used Neo4j graph database as the persistence storage. The main advantages we perceived are since we used graph theory concepts to evaluate microservice architecture, it is very useful to store services as nodes and dependencies among services as edges in a graph database. This allowed us to generate the dependency graph of the microservice system easily.

On the other hand, to generate the metrics such as Degree and Clustering coefficient, we need to run algorithms on the database. Neo4j provides a data science library which we can utilize to run the graph algorithms without we writing them on our own. For this research we mainly used centrality and community detection algorithms provided by the Neo4j library. The *centrality algorithm* we used was *degree centrality algorithm* to generate Degree related metrics and *triangle count algorithm* as a *community detection*

algorithm to generate *clustering coefficient* of services. If we went with a relational database none of the above could be achieved easily.

In the evaluation we were able to analyse the results provided by MAIG and identify the anti-patterns which we deliberately introduced in to the microservice system we developed. We saw that it is possible to use graph theory concepts to analyse a microservice system architecture when the services and dependencies are mapped to as a dependency graph.

6.2 Limitations

For the evaluation purposes we used a sample microservice system we developed. From that we can conclude that the five anti-patterns we focussed on could be accurately analysed and identified using graph theory concepts. However, to confirm the accuracy of the tool in general, the tool needs to be integrated with other production level microservice systems and evaluated. Also, the tool tracks only synchronous systems.

6.3 Future work

In this research we mainly focused on five service based anti-patterns that can be often found in microservice systems. We generated graph theory-based metrics specifically for these five anti-patterns. But there are many other services based anti-patterns that have been discussed in literature. As a future work we can extend the system such that it can identify more anti-patterns in a microservice system.

The Neo4j graph data science library provides many other algorithms as centrality and community detection algorithms. As an example, there are multiple other centrality algorithms such as *Page rank, Article rank, Betweenness* algorithms etc. Each of them can be used to gain different perspectives of a graph. We can look in to the possibility of applying those algorithms on the dependency graph of the microservice system and by that gain different insights of the microservice system.

The tool we developed is capable of tracking synchronous systems which communicate over HTTP. The metrics and visualizations are generated based on the synchronous behavior. For this research we haven't considered any asynchronous components such as message queues. As a future work we can extend the application to track asynchronous systems as well.

7 REFERENCES

[1] N. Alshuqayran, N. Ali, and R. Evans, "A Systematic Mapping Study in Microservice Architecture," *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 2016.

[2] J. Soldani, D. A. Tamburri, and W.-J. V. D. Heuvel, "The pains and gains of microservices: A Systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[3] R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger, "Principled workflow-centric tracing of distributed systems," *Proceedings of the Seventh ACM Symposium on Cloud Computing*, May 2016.

[4] B. Mayer and R. Weinreich, "An Approach to Extract the Architecture of Microservice-Based Software Systems," *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2018.

[5] S.-P. Ma, C.-Y. Fan, Y. Chuang, I.-H. Liu, and C.-W. Lan, "Graph-based and scenario-driven microservice analysis, retrieval, and testing," *Future Generation Computer Systems*, vol. 100, pp. 724–735, 2019.

[6] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," *Present and Ulterior Software Engineering*, pp. 195–216, 2017.

[7] "Microservices," *martinfowler.com*. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 21-May-2020].

[8] D. Taibi, V. Lenarduzzi, and C. Pahl, "Microservices Anti-patterns: A Taxonomy," *Microservices*, pp. 111–128, Dec. 2019.

[9] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[10] A. Rotem-Gal-Oz, *SOA Patterns*. Manning Publications, 2012.

[11] F. Palma and N. Mohay, "A study on the taxonomy of service antipatterns," *2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP)*, 2015.

[12] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[13] D. Hoyos, "Interactive Visualizations For Supporting The Analysis Of Distributed Services Utilization". *M.Sc. Technical University of Munich*, 2018.

[14] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. *Technical report, Google, Inc*.

[15] "Digraphs.," *Princeton University*. [Online]. Available: https://algs4.cs.princeton.edu/42digraph/. [Accessed: 21-May-2020].

[16] S. G. Shrinivas, S. Vetrivel, and N. M. Elango, "APPLICATIONS OF GRAPH THEORY IN COMPUTER SCIENCE AN OVERVIEW," *International Journal of Engineering Science and Technology*, vol. 2, pp. 4610–4621, Sep. 2010.

[17] "Network Science by Albert-László Barabási," *BarabásiLab*. [Online]. Available: http://networksciencebook.com/. [Accessed: 21-May-2020].

[18] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using Service Dependency Graph to Analyze and Test Microservices," *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 2018.

[19] R. Tarjan, "Depth-first search and linear graph algorithms," *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, 1971.

[20] Rahman, M. I., Panichella, S., & Taibi, D. (2019). A curated dataset of microservices-based systems. *In SSSME-2019: Joint Proceedings of the Inforte*

Summer School on Software Maintenance and Evolution (CEUR Workshop Proceedings; Vol. 2520). CEUR-WS.

[21] Neo4j Database. (2020, April 23). Retrieved May 23, 2020, from https://neo4j.com/neo4j-graph-database/

[22] Needham, M., & Hodler, A. E. (2019). *Graph algorithms: Practical examples in Apache Spark and Neo4j*. Beijing: O'Reilly.

[23] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In Pervasive computing and applications (ICPCA), *2011 6th international conference on*, pages 363{366. IEEE, 2011.

[24] Singh, M., Kaur, K.: Sql2neo: Moving health-care data from relational to graph databases. In: *2015 IEEE International on Advance Computing Conference (IACC)*, pp. 721–725. IEEE (2015)

[25] R. kumar Kaliyar, "Graph databases: A survey," *in Computing, Communication & Automation (ICCCA), 2015 International Conference on. IEEE*, 2015, pp. 785–790.

[26] Mark Graves, Ellen R Bergeman, and Charles B Lawrence. Graph database systems. *IEEE Engineering in Medicine and Biology Magazine*, 14(6):737{745, 1995.

[27] De Virgilio, R., Maccioni, A., Torlone, R.: Model-driven design of graph databases. In: Yu, E., Dobbie, G., Jarke, M., Purao, S. (eds.) ER 2014. LNCS, vol. 8824, pp. 172–185. Springer, Heidelberg (2014)

[28] Huang H, Dong Z (2013) Research on architecture and query performance based on distributed graph database neo4j. *In: Consumer Electronics, Communications and Networks (CECNet), 2013 3rd International Conference On. IEEE, Xianning*, China. pp 533–536

[29] G. Jaiswal and A.P. Agrawal, “Comparative Analysis of Relational and Graph Databases,” In IOSR Journal of Engineering (IOSRJEN) e-ISSN: 2250-3021, p-ISSN: 2278-8719 Vol. 3, Issue 8 (August. 2013), ||V2|| PP 25-27

[30] Mario Villamizar, Oscar Garces, Harold Castro, Mauricio Verano, ´ Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *In Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE, 2015.

[31] Hasselbring, W., Steinacker, G.: Microservice architectures for scalability, agility and reliability in e-commerce. *In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 243–246 (2017)

[32] Granchelli, G., Cardarelli, M., Francesco, P.D., Malavolta, I., Iovino, L., Salle, A.D.: Towards recovering the software architecture of microservice-based systems. *In: 2017 IEEE International Conference on Software ArchitectureWorkshops (ICSAW)*, pp. 46–53 (2017)

[33] Bogner, J.,Wagner, S., Zimmermann, A.: Towards a practical maintainability quality model for service-and microservice-based systems. *In: Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, ECSA 2017*, pp. 195–198. ACM, New York (2017)

[34] Pei Breivold, H., Crnkovic, I., Eriksson, P. J.: Analyzing Software Evolvability. *Accepted at 32nd COMPSAC*. (2008)

[35] Kalske, M., Mkitalo, N., Mikkonen, T.: Challenges when moving from monolith to microservice architecture. *In: Current Trends in Web Engineering, pp. 32–47. Springer, Cham* (2017)

[36] Richardson, C.: Microservices—Pattern: Microservice Architecture, March 2014. http://microservices.io/patterns/microservices.html

[37] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Gueh´ eneuc, ´ B. Baudry, and J.-M. Jez´ equel, “Specification and detection of soa antipatterns” *in Service-Oriented Computing*. Springer, 2012, pp. 1–16.

[38] B. Shim, S. Choue, S. Kim, S. Park. “A Design Quality Model for Service-Oriented Architecture.” *In: 2008 15th Asia-Pacific Software Engineering Conference. IEEE,* 2008.

[39] T. Senivongse, A. Puapolthep. “A Maintainability Assessment Model for Service-Oriented Systems.” *In: Proceedings of the World Congress on Engineering and Computer Science. San Francisco, CA, USA: Newswood Limited,* 2015.

[40] N. M. Nik Daud, W. M.N. Wan Kadir. “Static and dynamic classifications for SOA structural attributes metrics.” *In: 2014 8th. Malaysian Software Engineering Conference (MySEC). Langkawi, Malaysia: IEEE,* Sept. 2014.

[41] T. Engel, M. Langermeier, B. Bauer, A. Hofmann. “Evaluation of Microservice Architectures: A Metric and Tool-Based Approach.” Cham: Springer International Publishing, 2018.

[42] D. Taibi, K. Systä. “From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining.” *In: Proceedings of the 9th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications,* 2019.

[43] Akoglu, L., Tong, H., Koutra, D.: Graph based anomaly detection and description: a survey. *Data Min. Knowl. Disc.* ***29****(3), 626–688* (2014).

[44] F. Palma, L. An, F. Khomh, N. Moha, Y.-G. Gueheneuc. “Investigating the Change-Proneness of Service Patterns and Antipatterns.” *In: 2014 IEEE 7th International Conference on Service-Oriented Computing and Applications. IEEE,* Nov. 2014