PERFORMANCE FORECASTING FOR WEB SERVICES IN CLOUD ENVIRONMENT

Clément Cassé

▶ To cite this version:

Clément Cassé. PERFORMANCE FORECASTING FOR WEB SERVICES IN CLOUD ENVIRONMENT. Networking and Internet Architecture [cs.NI]. UPS Toulouse, 2023. English. NNT : 2023TOU30268 . tel-04544207v1

HAL Id: tel-04544207
https://laas.hal.science/tel-04544207v1

Submitted on 29 Jan 2024 (v1), last revised 12 Apr 2024 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *26/10/2023* **par :**

Clément CASSÉ

PRÉVISION DES PERFORMANCES DES SERVICES WEB EN ENVIRONNEMENT CLOUD

JURY

Pierre SENS	Professeur	Président du Jury
Géraldine TEXIER	Professeure	Rapporteure
Yves ROUDIER	Professeur	Rapporteur
Gilles TREDAN	Chargé de Recherche	Examinateur
Philippe OWEZARSKI	Directeur de Recherche	Directeur de Thèse
Pascal BERTHOU	Professeur	Co-Directeur de Thèse
Sébastien JOSSET	Ingénieur	Invité

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
LAAS - Laboratoire d'Analyse et d'Architecture des Systèmes
Directeur(s) de Thèse :
Philippe OWEZARSKI et *Pascal BERTHOU*
Rapporteurs :
Géraldine TEXIER et *Yves ROUDIER*

Acknowledgments

Avant toute chose je tenais à adresser mes remerciements à toutes les personnes qui ont pris part à l'encadrement de la thèse. En tout premier lieu Philippe et Pascal pour l'encadrement académique: je vous remercie de vos conseils qui m'ont permis de m'approprier et de développer ce sujet qui me tenait à coeur. Vos visions et vos expertises m'ont apporté une aide précieuse pour structurer, organiser et présenter mon avancement et mes idées. J'espère pouvoir tirer parti de ces enseignements même au-delà du monde de la recherche.

De plus, je remercie aussi Sébastien JOSSET pour avoir rendu ce contexte industriel possible en me permettant d'intégrer Orange et sa division de recherche et développement. Intégrer différentes équipes de développement d'application a été l'occasion pour moi de me familiariser avec plusieurs problématiques réelles quant à la conception d'application cloud.

Je remercie également Géraldine TEXIER et Yves ROUDIER d'avoir accepté d'être rapporteurs pour cette thèse ainsi que Pierre SENS et Gilles TREDAN d'en être examinateur.

J'exprime ma gratitude auprès des différentes équipes de développement à Orange où toutes m'ont bien accueilli et fourni des informations précieuses pour mon état de l'art ainsi que pour bien affiner ma problématique de recherche. Je remercie en particulier les équipes de Djingo, et en particulier Charle-Henri GIDEL et Sébastien DELEPLACE, qui m'ont accueilli dans leur équipe, ce qui a grandement contribué à justifier mon approche. C'est dans cette équipe que j'ai pu développer et tester mon modèle pour répondre à des problèmes concrets rencontrés par des développeurs d'applications cloud.

En revanche, je ne remercie pas du tout la pandémie de COVID qui a eu lieu pendant cette thèse, ni les joyeusetés qui y ont été associées telle que les confinements et les interactions sociales à travers *d'applications Web hébergées en environnement cloud.* Mais aussi le recentrage de l'entreprise Orange autour d'autres axes de développement qui a conduit à limiter les développements sur Djingo avant que je puisse tester mon modèle dans des conditions réelles.

C'est donc le moment d'adresser mes remerciements à tous mes amis et collègues du LAAS qui ont partagé avec moi cette période de questionnement, particulièrement David, Guillaume, Flore, François, Lucian, Quentin, Alex, Laurent, Tanissia, Imane, Stan, ... Enfin, j'adresse mes derniers remerciements à ma famille et surtout à mes parents pour leur soutien sans failles tout au long de cette aventure. De la candidature jusqu'à la soutenance, votre regard bienveillant m'a encouragé à donner le meilleur de moi même.

Contents

Introduction **1**
Cloud Native Applications . . . 1
Industrial Context & Problem . . . 3
Contributions & Dissertation Outline . . . 4

1 Cloud Application Performance Monitoring **7**
1.1 Introduction . . . 7
1.2 The Cloud Computing Paradigm . . . 9
1.2.1 Overview and Definition of Cloud Computing . . . 9
1.2.2 Evolution of Cloud Computing . . . 12
1.2.3 Performance Measurements and Concerns . . . 15
1.2.4 Closing Words on the Cloud Paradigm . . . 17
1.3 Cloud-Native Application Architecture . . . 17
1.3.1 The Microservices Architecture . . . 18
1.3.2 The Evolution of Cloud Resource Management . . . 22
1.3.3 Challenges Regarding Performance Evaluation . . . 24
1.4 Cloud Application Monitoring . . . 24
1.4.1 Cloud Application Monitoring and Performance Analysis . . 24
1.4.2 Toward a Unified Cloud Application Monitoring Framework . 27
1.4.3 Research Challenges for Monitoring Cloud Application 28
1.5 Scope of the Thesis . . . 30
1.6 Conclusions . . . 31

2 A Hierarchical Property Graph Model **33**
2.1 Introduction . . . 33
2.2 Distributed Tracing Ecosystem . . . 35
2.2.1 OpenTelemetry Data Collection Architecture . . . 35
2.2.2 Collecting Traces in Jaeger Tracing . . . 37
2.2.3 The Jaeger Analytic Library and its Limitations . . . 38
2.3 Extracting a Structural Model from Traces . . . 39
2.3.1 Identifying Common Elements to Aggregate Traces . . . 39
2.3.2 Modelling an Application Hierarchical Structure . . . 41
2.4 Modelling Components Interactions . . . 43
2.4.1 Leveraging the Property Graph Model to Identify the Type of Communication . . . 43
2.4.2 Graph Rewriting Operations . . . 45
2.4.3 Building a Hierarchical Property Graph . . . 49
2.5 Implementation . . . 50
2.5.1 Extracting Data from a Jaeger gRPC Endpoint . . . 51
2.5.2 Property Graph Encoding . . . 52
2.5.3 Graph Rewriting Operations . . . 55
2.6 Conclusions . . . 57

3 Identifying Inefficient Service Composition 59
3.1 Introduction . . . 59
3.2 Modelling a System With Hierarchies . . . 61
3.2.1 Definition and Subtypes of Hierarchy . . . 61
3.2.2 Measuring Imperfect Flow Hierarchies . . . 62
3.2.3 Cycle Identification . . . 64
3.3 Detecting Inefficient Service Composition . . . 66
3.3.1 Application to the Hierarchical Property Graph . . . 66
3.3.2 Proof of Work on a Sample Cloud Application . . . 68
3.4 Implementation . . . 71
3.4.1 Designing a Multi Layers Platform with Zonal Kubernetes Cluster . . . 71
3.4.2 Getting OpenTelemetry Traces With Network Level . . . 72
3.4.3 Computing the Flow Hierarchy Metric . . . 75
3.4.4 Results . . . 77
3.5 Conclusions . . . 79

4 Identifying Bottlenecks with Centrality 81
4.1 Introduction . . . 81
4.2 Generalizing the Graph Encoding Model . . . 83
4.2.1 Including Multiple Resource Type in the Model . . . 83
4.2.2 Configuring the Containment Hierarchy . . . 85
4.2.3 Characterizing an AWS Application . . . 87
4.3 Application to Complex Cloud Applications . . . 88
4.3.1 Overview of Graph Centrality Algorithm . . . 89
4.3.2 Distributed Applications Bottlenecks . . . 92
4.4 Implementation . . . 93
4.4.1 Using Spigo for Emitting OpenTelemetry Traces . . . 94
4.4.2 Scenario Selection and Representativeness . . . 95
4.4.3 Observing the Impact of Betweenness Centrality in the Riak Simulation . . . 97
4.5 Conclusions . . . 101

Conclusion & Future Works 103
Synthesis of Contributions . . . 104
Future Works . . . 105
Short-term Work . . . 106
Mid-term Work . . . 108
Long-term Work . . . 109
Closing Words . . . 109

A Résumé en Français 111
A.1 Monitoring d'Applications Cloud . . . 114
A.1.1 Introduction . . . 114
A.1.2 Présentation du Paradigme Cloud . . . 114
A.1.3 Les Applications Cloud Natives . . . 115

A.1.4 Monitoring et Analyse de Performance d'Application cloud . 116
A.2 Modélisation des Communications Internes 118
A.2.1 Introduction 118
A.2.2 Présentation de l'Écosystème de Tracing 118
A.2.3 Extraction et Aggregation des Données dans un Graphe de Propriétés 119
A.3 Détection de Composition de Services Inefficaces 123
A.3.1 Introduction 123
A.3.2 Modélisation d'une Application cloud grâce au Concept de Hiérarchies 123
A.3.3 Détection de Communications Inefficaces 124
A.3.4 Mise en œuvre 126
A.4 Détection de Goulots d'Étranglements 128
A.4.1 Introduction 128
A.4.2 Généralisation de l'Encodage en Graphe de Propriétés 128
A.4.3 Utilisation de l'analyse de Centralité pour l'Anticipation de Goulots d'Étranglement 130
A.4.4 Vérification Expérimentale 132
A.5 Conclusion 137
A.5.1 Synthèse des Contributions 137
A.5.2 Pistes de Poursuite des Travaux 138

B Scala Notebook and Code 141
B.1 Data Aquisition 141
B.1.1 Establishing a Channel With a Jaeger Instance 141
B.1.2 Mapping ProtoBuf Data to Standard Java/Scala API 143
B.2 Definition of the Analytics Trace-Data-Model 146
B.2.1 Operation Entities 146
B.2.2 Resources Entities 146
B.2.3 Span Entities 147
B.2.4 Trace Entities 148
B.3 Graph Encoding 149
B.3.1 Defining the Property Graph Model 149
B.3.2 Encoding Process 150
B.3.3 Graph Rewriting 152
B.4 Calculation of the Flow hierarchy metric 153
B.5 Building and Running the Pipeline 155

Bibliography 159

List of Figures

1 *Google Cloud Platform* Data Centre and Network 2

1.1 Representation of abstraction levels with virtual machines and with containers. Illustration from [Bistarelli 2018] 13
1.2 Amazon and Netflix “Death Stars” from 2016 21

2.1 OpenTelemetry Pipeline Architecture 35
2.2 Part of a Jaeger Trace from an Online Boutique Application 37
2.3 Graphical Representation of Jaeger Analytic Library Graph Encoding 38
2.4 Transforming a Trace in a Property Graph. 40
2.5 Transforming a Trace in a Property Graph. 42
2.6 Example of Local Communications and Network Communication Between Resources 44
2.7 Graph Pattern Showing a Local Reference 44
2.8 Graph Pattern Showing a Network Reference 44
2.9 Graph Rewriting Approach to deduce resources dependencies (applied to Pods) based on a Simple Pushout operation. 47
2.10 Graph Rewriting process explained step by step 48
2.11 Visual Pattern representing equation 2.1 49
2.12 Hierarchical Graph Representation. 50
2.13 Complete Telemetry Processing Pipeline 51
2.14 Aggregation of Multiple Traces Within a Single Graph Instance . . . 55
2.15 Rewriting Operation on the Graph at the *Pods* level 57

3.1 Graph Representing Communications Between Components of a Distributed Application 63
3.2 Focus on Portions of the Graph: With Clearly Identifiable Hierarchy Levels (left) and With No Identifiable Order of Vertices (right) . . . 64
3.3 Illustration of M_d matrix computation from [Luo 2011] where M^n is the link adjacency matrix raised to the power n 65
3.4 Example of Vertices Grouped by Strongly Connected Components (SCC) in a Sample Directed Graph 65
3.5 Examples of Flow Hierarchy Metric Calculation at Each Layer of the Containment Hierarchy for Two Traces 67
3.6 Example Where All Pods Are Executed on the Same Node 68
3.7 Diagram of the Microservices Demo Application with Components Interactions Provided by the documentation of the Application . . . 69
3.8 Graph Transformation for a Particular Trace 70
3.9 Illustration of a Normal Kubernetes Cluster (on the left) and a Zonal Kubernetes Cluster (on the right) 71
3.10 Complete Telemetry Pipeline 73
3.11 Example of the Network Communications Reported in a Trace . . . 74

3.12 Graphical representation of traces and their flow hierarchy metric ($h_{Pods}, h_{Nodes}, h_{Zones}$) depending on the end-to-end response time of the trace 78

4.1 Resource Base-Graph for a Generic Meta Model 85
4.2 Example of adding the containment of resources for the embedding $Pod \subset Node \subset Zone \subset Region$. 86
4.3 Meta Model of the graph encoding applied to traces for AWS Applications 87
4.4 OpenTelemetry graph encoding pipeline 88
4.5 Sample Graph Exhibiting Different Topologies. 90
4.6 Logical architecture of AWS components in the Riak Scenario 95
4.7 A sample of the graph of services with a focus in *ingestMQ* services . 97
4.8 Cloud Application Graph Visualization Where Vertices Size is Proportional to the Betweenness Centrality Score of the Vertex 98
4.9 Cloud Application Graph Visualization respectively with one and five instances of *ingestMQ* per zones 100
5.1 Behaviour Principle of the Hierarchical Edge Bundling Visualization: the Tension of the Arcs Depend on the Number of Layers Traversed 107
5.2 Example of Edge Bundling Visualization provided by *infra-scrapper* to represent Pod Interaction in a Kubernetes Namespace 108

A.1 Visualisation sous forme de Graphe des Microservices des applications de Amazon et Netflix en 2016 par [Cockcroft 2016a, Cockcroft 2016b].116
A.2 Graphical Representation of Jaeger Analytic Library Graph Encoding 119
A.3 Processus d'encodage d'une trace en un graphe de propriétés. 120
A.4 Représentation des motifs de communication locale et de communication réseau dans notre modèle d'encodage de graphe. 121
A.5 Formalisation du processus de réécriture de graphe 121
A.6 Trace encodée représentée sous forme de graphe hiérarchique 124
A.7 Identification des composants fortement connectés dans un graphe dirigé 125
A.8 Calcul d'indice de flow hierarchy sur deux placements de Pods dans un Cluster Kubernetes 126
A.9 Décomposition du modèle d'encodage de graphe en deux étapes pour les relations *EXECUTES_ON* et *IS_CONTAINED*. 129
A.10 Méta Modèle du processus d'encodage de graphe après les étapes d'extraction et de sélection des relations. 129
A.11 Graphe mettant en avant différentes topologies pour illustrer les différents algorithmes de centralité. 130
A.12 Architecture logique des composants formant le scénario *Riak* du Simulateur Spigo 133
A.13 Visualisation du scénario *Riak* de *Spigo* avec trois instances de *ingestMQ* par zones dans laquelle sommets du graphe sont d'autant plus gros que leur indice de centralité intermédiaire est élevé. 134

A.14 Visualisation des graphes après calcul de l'indice de centralité d'intermédiarité pour les cas où le nombre d'instances de *ingestMQ* est respectivement de 1 et de 5 par zones de disponibilités 135
A.15 Visualisation *Edge Bundling* matérialisant la hiérarchie d'inclusion par une tension des arcs plus forte vers le centre quand le nombre de couches hiérarchiques traversées est élevé. 139

List of Tables

2.1 OpenTelemetry Cloud Semantic for Resource Location 36

4.1 OpenTelemetry Cloud Semantic for Graph Encoding 84
4.2 Legend of AWS symbols 96
4.3 Betweenness Centrality score range for each group of services 99
4.4 Betweenness centrality score range for each group of services with a varying number of *ingestMQ* instances 99

A.1 Score de centralité intermédiaire associé à chacun des services de l'application cloud simulée *Riak* pour les cas ou le nombre d'instances du service *ingestMQ* est respectivement de 1, 3, et de 5. 136

Introduction

Over the last decade, we observed an exponential growth of the number of Web applications covering a wide spectrum of usages. Social-Networks, Online Shopping and Multimedia Streaming platforms are just a few examples of the services available today via these Web applications. And still today, the number of internet managed services and applications keeps on increasing on a daily basis. This environment becomes extremely competitive; therefore, companies have to keep their user-base engaged in their products. This user engagement is key for their business and is maintained by a high quality of service. Indeed, over the past years, many feedbacks from large-scale companies demonstrated that the application response time was a driving metric for estimating user engagement. Back in 2006, Google made an experiment that caused the response time of its search engine to be increased by 500 ms, this extra delay caused the traffic of the search engine to drop by 20% [Souders 2009, Ibidunmoye 2015]. Also, in 2008, Amazon declared that each 100 ms added to their latency reduced their sales by 1% [Ibidunmoye 2015]. Therefore, monitoring and optimizing a Web application performance became an important part of developers' job.

In addition, a recent study on the impact of COVID-19 on internet traffic showed that Youtube traffic represented more than 15% of all the global internet traffic. Also, during the lockdown, the part of Netflix on the global traffic rose up to 11%, becoming the second most important website in terms of global traffic behind YouTube [Cantor 2020]. To be able to sustain such a tremendous load, these applications have been designed to adapt to the number of users while maintaining an almost 100% availability and keeping their response time as low as possible. Although, not all Web-Application requires the extreme scalability of Youtube or Netflix; these recent events highlighted that Web Applications are now designed to manage a quickly evolving load of requests.

The main enabler of this extreme scalability has been the Cloud Computing paradigm. By leveraging virtualization and automation technologies, the Cloud Computing allows offloading the management of some components to Third-Party companies. Developers can now focus on creating business-centric code and leave the complex and time consuming tasks of operational management to Cloud Providers.

Cloud Native Applications

Cloud Computing is a complex concept involving an ever-increasing spectrum of technologies, concepts and challenges. Still, this paradigm has been widely adopted in the recent years and has changed the way software is designed and developed nowadays. To keep these web applications performant and scalable, this approach shifts away from the traditional definition of a computer program and defines a software as a heavily distributed system. Today, the most popular applications have adopted this change in design, they are now made of specialized business-centric smaller components dedicated to one task and communicating with each other over

the network. These applications are called Cloud Native Applications. Also they take full benefits from the global distribution of the Cloud provider around the world to replicate their applications on each continent as close to the user as possible. Figure 1 shows the data centres owned by Google for their *Google Cloud Platform* public Cloud. It also shows the network links dedicated to the platform and owned by the company. Cloud Native Applications leverage this global distribution of data centres get better performance.

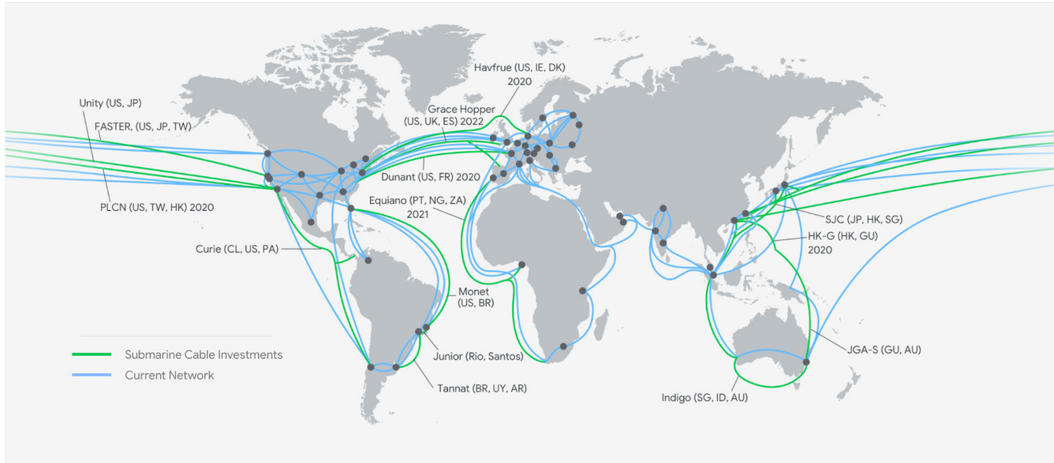

Figure 1: *Google Cloud Platform* Data Centre and Network

Although, by embracing a distributed architecture, both software debugging and performance analysis became a more complex problem. Indeed, whereas detecting software bugs and hardware failures still take an important part in Cloud-Native Application monitoring, there is also a wide variety of issues related to distributed systems. At higher scale, an application performance can also be affected by the network, or by the multiple virtualization layers involved, or even by other processes executed in the same data centre. For Cloud-Native Application monitoring, this results in new problems to detect and address:

1. Detection integration error and misconfiguration of the components of the system. These misconfigurations may result in cascading errors or inefficient service composition.

2. Detection of bottlenecks in the system. When an application has to undergo an increased load, the load of each component in the system may not be distributed evenly; some components may contribute to a greater extent to the overall application response.

3. Detection of noisy neighbours or resource vampyrisation. Cloud environments notoriously rely on resource over-commitment in their data centre, also while designing a distributed application, over-commitment of computing resources can also be a design strategy for cost optimization.

Whereas Cloud Native Application Monitoring has not followed the evolution pace of Cloud Computing in general, there are still initiatives aiming to address these new challenges. *OpenTelemetry* is an open source project supported by the most prominent Cloud-Computing companies like Google, Microsoft or Amazon. It aims to standardize monitoring for Cloud-Native Application. At the time of writing, the project was rather young and not suitable for a production usage, still it had some promising building blocks we will focus on this thesis. OpenTelemetry focus on normalizing the semantic and the format of telemetry data in Cloud environment. The main contribution of OpenTelemetry has been to normalize distributed tracing. Distributed Tracing is a new kind of instrumentation introduced in 2010 by Google that is capable of following the propagation of a request in a distributed system and provide a high-level view of the components interaction for a given request. With distributed tracing, monitoring data is not only a collection of multi-dimensional points lying in space independently; on the contrary distributed tracing focuses on exhibiting inter dependencies between measurements.

Given this context of open challenges for Cloud Native monitoring and this new monitoring data-structure, the focus on this thesis has been to leverage traces to address the issues of distributed systems.

Industrial Context & Problem

The work presented in this thesis has been part of a research project initiated by Orange Labs Services, the research and development division of the Orange company. This work has been motivated by the increased adoption of Cloud technologies to design the company services. Recently, Orange started developing *Djingo*, a voice assistant application hosted in multiple cloud data centres spread out across Europe. By their nature, voice assistant applications require a high level of performance to provide feedback to the user in a natural way. Multiple Cloud Service Providers have been involved for deploying the Application, but the decision by the company has been to avoid relying on resource over commitment. Still to maintain a low response time, the detection of integration errors and misconfigurations as well as bottlenecks was a crucial part.

During development and tests of the application, scattered on multiple data centres, there was a need to investigate deeper the cause of some latency issues that were observed. For this reason, the company integrated tracing to its product, to help investigate on latency issues. Still the amount of data generated by tracing instrumentation was too consequent to be analysed and required further processing. Indeed, even with tracing, there was no immediate way of identifying problematic behaviours and estimate their impact on the overall performance of the application. In particular, with the automatic allocation of microservices in Cloud platforms, there was a need of identifying the impact of the communications crossing data centre boundaries. And while, these communications cannot be completely avoided, they can, sometime, be reduced by opting for local component instead of distant ones.

The problem of detecting Cloud Application specific performance issues has

been tackled down under the angle of modelling a global view of a distributed application. Indeed, unlike standalone applications which can easily get the global view of the system, distributed applications communicate over the network and hide their underlying implementations. A monitoring technique capable of maintaining a global application model at runtime would provide insight on the communications of the different entities of the system and instrument a part, often disregarded of application monitoring: the network.

Contributions & Dissertation Outline

In this thesis we propose to use the recent Distributed Tracing data that is being normalized with the OpenTelemetry initiative. As of today, traces are used by developers to debug their Cloud Applications but almost never on an application running in production. With this thesis, we propose a model maintained at runtime by traces that aims to maintain a global view of a physically distributed Cloud Application. Then we leverage this model to address distributed systems common issues that are traditionally encountered in Cloud Applications: the identification of inefficient service placement and bottlenecks identification.

The contribution of this thesis is organized as follows:

Chapter 1: Cloud Application Performance Monitoring: This chapter presents Cloud Computing in general, it provides a literature review of the key concept that will be used throughout this thesis. An analysis is provided by putting into perspective widely adopted academic works and definitions with the State-of-the-Art Cloud technologies. Also, an emphasis is made on Cloud Application structure and its impact on performance assessment. Then a review is made on the up-to-date monitoring techniques used both by academics and the industry to assess performance in a Cloud environment. This study aims to provide the context behind the use of distributed tracing, an emergent technology used in later chapters of this thesis.

Chapter 2: A Hierarchical Property Graph Model: In this chapter, a focus is made on distributed tracing data in general. We present the actual ecosystem for obtaining and processing traces on a Cloud Application. Then, by leveraging the associated semantic, a generic model is detailed that focuses on exhibiting the location of resources within a graph. The resulting graph represents components of a Cloud application located in multiple data centres, and models them as a hierarchical property graph. A method that encodes multiple traces is covered as well as its implementation working on an online flow of traces. The work presented in this chapter corresponds to the common grounding enabling both the detection of inefficient service composition and bottlenecks, the topics covered in later chapters.

Chapter 3: Identifying Inefficient Service Composition: The focus of this chapter is set on the interactions of components within the layers of the hierarchical property graph. Communications among entities of a layer may

exhibit patterns that translates into either inefficient placement or misconfigurations. This chapter focuses on the identification of cycles among higher-level resources, which translates in an inefficient resource composition. A platform, hosted in a real cloud platform, has been deployed. It runs Kubernetes but has a layered networking model, in our experiments, it has been shown that Kubernetes does not provide an efficient resource composition, and in most cases opts for a costly composition of services.

This work has been published in [Cassé 2021].

Chapter 4: Identifying Bottlenecks with Centrality This chapter focuses on a global analysis of the application performance model provided in chapter 2. This chapter presents a slighlty different approach to consume tracing data. While the flow hierarchy metric was computed online on every individual traces, this chapter proposes an approach to analyse the global model to identify hotpoints in the system. In this chapter, an offline analysis is presented to detect services involved on many requests and service compositions that act as bottlenecks for an application. Bottlenecks are identified with graph centrality algorithms. The work presented in this chapter is tested on a simulation application that implements several scenarios based on state-of-the-art Amazon Web Services architectures.

This work has been published in [Cassé 2022].

Finally, the **Conclusion & Future Works** chapter closes the thesis by providing a summary of the contributions and opening on future works.

Chapter 1

Cloud Application Performance Monitoring

Contents

1.1 Introduction **7**
1.2 The Cloud Computing Paradigm **9**
1.2.1 Overview and Definition of Cloud Computing 9
1.2.2 Evolution of Cloud Computing 12
1.2.3 Performance Measurements and Concerns 15
1.2.4 Closing Words on the Cloud Paradigm 17
1.3 Cloud-Native Application Architecture **17**
1.3.1 The Microservices Architecture 18
1.3.2 The Evolution of Cloud Resource Management 22
1.3.3 Challenges Regarding Performance Evaluation 24
1.4 Cloud Application Monitoring **24**
1.4.1 Cloud Application Monitoring and Performance Analysis . . 24
1.4.2 Toward a Unified Cloud Application Monitoring Framework . 27
1.4.3 Research Challenges for Monitoring Cloud Application 28
1.5 Scope of the Thesis **30**
1.6 Conclusions **31**

1.1 Introduction

The concept of *Cloud Computing* emerged more than a decade ago; it qualifies the idea of outsourcing computing resources into dedicated data centre managed by third-party companies. Since 2008, the adoption of *Cloud Technologies* by companies has been massive at the point to now become the *de facto* model for designing large-scale Web Applications. Nowadays, Cloud Computing is perceived as the enabler of many other well-rooted theories in the field of Information Technologies: Infrastructure automation, microservice architecture, or processing pipelines used in Big Data have all been empowered by Cloud technologies.

The most prominent American technology companies: Facebook, Amazon, Apple, Microsoft and Google, have all a part of their business now related to the *Cloud*. And, whereas Facebook and Apple build and use their own data centres for their internal usage, Amazon, Google and Microsoft have gone a step further by reselling

their computing power to other companies. They created their *Public Clouds*, respectively named Amazon Web Services (AWS), Google Cloud Engine (GCE) and Azure. These companies exposed some of their expertise through a catalogue of services hosted in their data centres and available for all developers to use with attractive pricing. Now, other companies providing similar offers appeared, they are called Cloud Service Providers (CSPs).

Cloud Computing made far-reaching changes in the approach of resource provisioning. By providing means to automate and dynamically scale application components, this technology distorted the traditional software infrastructure. The adoption of this paradigm is such that today numerous companies only rely to CSP to host their entire infrastructure; as Netflix pioneered back in time to rely on AWS to host their massive application.

While the economic benefits of *Cloud Technologies* are no longer to be demonstrated, adopting this paradigm is a tremendous gap in terms of software design: Cloud Native Applications are built by integrating third-party components with business-centric code [Heinrich 2016]. A resulting side effect is that *Cloud Computing Software Design* also shakes the foundations of software monitoring. Cloud Applications are designed to act as distributed systems. Their underlying infrastructure is delivered *as-a-Service*: components are perceived as a black box and the Quality of Service (QoS) remains one of the few metrics available.

In the following chapter, we will present a literature study presenting the Cloud Computing and its evolution over the past years. Indeed, Cloud technologies have evolved at a rapid pace over the last two decades motivated by an ever-increasing industrial need. This chapter aims to set some steps back and to present the trends that motivated this evolution. We put these trends in perspective with the impact they had on system monitoring to identify the key challenges of Cloud-Application performance analysis. Throughout our literature review, the term *Cloud Computing* has taken multiple meanings: Often presented as a technology specific to data centres based on multiple computing and networking virtualization techniques, it evolved as framework helping developers to create distributed applications. Also, initially presented for its centralization capabilities, the most recent challenges have been to be extendable to multiple places around the world.

The following study is driven and supported by the performances challenges and questions raised when developing *Djingo*, a Cloud application designed at Orange. *Djingo* was designed to be a voice assistant that leveraged multiple data centres to be as close as possible to the end user, so that this would reduce the response time and aim to seamless interactions. Therefore, in this literature review, we first consider Cloud-Computing as a technology. We focus on the original definition and we set it in perspective with the technology that is actually in use to highlight the key challenges of performance modelling. Then we shift our point of view to consider Cloud computing as a framework to design highly scalable distributed application applications. We address the microservices architecture and infrastructure automation, then a focus is set on their impact on resource management. These changes lead to design applications as distributed systems instead to design them as standalone systems. So, we finally present research and industrial initiatives that tackle distributed system monitoring in the context of Cloud Computing. That led

to the recent unified monitoring framework for Cloud Application that will support our work throughout this thesis: *OpenTelemetry*.

The study is organized as follow:

Section 1.2 discusses the concept of *Cloud Computing* as a whole and provides a state-of-the-art of this engineering-heavy topic. We start from the U.S. National Institute of Standards and Technologies (NIST) definition and go toward the main evolution observed through the last decade. In particular, this section includes the adoption of containerization, a technology that is now omnipresent in Cloud-related topics. We also provide a review of other notable evolutions of the *Cloud* from the NIST definition like the concept of FaaS or Edge-Computing to finally land on the performance aspect.

Section 1.3 focuses on a complementary aspect of *Cloud Computing*: Software design of Cloud-Native Applications. Indeed, as Cloud Technologies shacked the foundations of software design by providing entirely new way of deploying applications, upper layers have therefore been affected. Cloud-Native Software now opposes to Monolith Applications, they are made of a multitude of components interacting with each other over the network. By taking the shape of distributed systems, so-called *Cloud-Native* Application became more difficult to monitor; this section reviews these new challenges.

Section 1.4 puts a focus on the advances and actual challenges of monitoring in such an application. Starting from real-world observations from prominent industrial, we generalize these feedbacks and identify the goal of monitoring in this context. This section is put into perspective with the open source initiative named *OpenTelemetry*.

Section 1.5 sums up the research challenges presented in this section and present the positioning of these work regarding these open research challenges.

Section 1.6 Concludes the background study and opens toward the contribution of the thesis.

1.2 The Cloud Computing Paradigm

1.2.1 Overview and Definition of Cloud Computing

The term *Cloud Computing* emerged as the *buzzword* that flooded commercial speech of the brand-new services provided by IT companies these last decades. While the term is very vague and gave rise to many interpretations, some more formal definitions have been proposed to provide a better understanding of this concept [Vaquero 2008, Mell 2011]. Even if there is still no consensus on the definition, the general idea remains the same: *Cloud Computing* allows outsourcing all kinds of computing resources to third-party companies that will be managed in their own data centre. These companies are called Cloud Service Providers (CSPs), they provide an immediate network access to these resources following a *pay-as-you-use* model.

The definition provided by the U.S. National Institute of Standards and Technologies (NIST) remains the most accepted among both academics and industry:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models. ([Mell 2011])

The NIST definition also highlights the following essential characteristics shared by Cloud implementations:

On-demand self-service Resource provisioning does not require any human interactions in the process. Most cloud providers expose an API to allow customers to provision or release resources in an automatic way.

Broad network access Once created, the computing resources are exposed to the network and can be reached via traditional remote control mechanisms.

Resource pooling Cloud providers are able to serve multiple customers while providing them virtually separated resources. They rely on virtualization to segregate their pool of resources into smaller subgroups, thus enabling multitenancy. The resource allocation made by the hypervisor is obfuscated to the final user, giving it only minimal information on its virtualized resource.

Rapid elasticity With the pay-as-you-use model and the on-demand self-service, Cloud customers can create their own logical component that adjusts, at runtime, the amount of resources they subscribe to.

Measured service Cloud providers basically give access to their resources *as-a-Service*. The Quality of Service (QoS) is, as a result, a metric driving both providers and customers in order to qualify whether or not the services match the expectations in terms of performance.

This definition also presents two complementary models: the deployment model and the service model. These two models help to define the separation of the scope of the CSP from the scope of the client. The deployment model describes the management of the underlying resources used by CSPs as well as who they target as customers. In another hand, the service model provides a classification of the various kinds of services provided to the users by CSPs.

The deployment model details the following management of resources:

Private Cloud: They target exclusively customers within the organization they are deployed on. They are often deployed on-premise but it's not mandatory and host Applications that should not be exposed over the internet. The goal of building a private cloud for an organization is to propose a common base, managed by a dedicated team, to be used by multiple projects within the organization.

Community Cloud: They target a wider audience than the private Clouds but still remains a closed audience. The communities are usually organizations or working groups that share common concerns like compliance or security constraints. Community Clouds are Cloud implementations that usually satisfy these constraints.

Public Cloud: They target the general public and are available on all the internet. There are multiple public Cloud providers. However Amazon, Google and Microsoft are leader on the market of Public Cloud. Their customers are usually developers wanting to host a public web application.

Hybrid Cloud: They are a composition of several of the previous deployment models while still being identifiable as a unique entity.

This definition heavily revolves around the separation of the client, consuming resources, from a third party, fully dedicated to the management of these services. It contrasts to the original software management in that it requires a company to manage the full stack from physical machines to business-centric code. The consequences are that Cloud computing introduced strong and opaque boundaries between the Cloud Providers and the client. And, while it eases creating and deploying software, these strong abstraction layers obscure debugging and make performance analysis an extremely complex task.

In the following, we call a service a contract materializing this opaque abstraction layer separating the provider from the client. And, in Cloud Providers catalog, services are many; to clearly identify the scope of these services, the NIST definition provides the service model:

Infrastructure as a Service (IaaS) heavily revolves around the concept of virtualization, it allows a CSP to separate resources in its data centre into smaller units, each rented to different clients. These resources can be Virtual Machines but also virtual network equipment or storage units.

Platform as a Service (PaaS) abstracts further the underlying system by providing already tuned up system capable of executing code. For example, PaaS can provide Python, Java, PHP, Ruby or Node.js runtimes. By using a PaaS developers do not have to install and manage these runtimes and rely on the provider to get a service running immediately. PaaS catalogue also includes common components like Databases or Caches. However, by relying on a third party to manage underlying components the developer cannot access lower-level metrics like the memory used, or the CPU time.

Software as a Service (SaaS) is the last abstraction layer: it allows the developer to use existing domain-specific software part of their application. SaaS usually covers pieces of software that are common to many applications, this includes user management, a search engine, a monitoring platform or a Payment solution.

This model acts as a reference to the concept of Cloud Computing: it describes a centralized model focused around CSPs data centres. The Cloud Provider, by relying on virtualization technology, subdivides its data centre into virtual resources to rent them to multiple clients, therefore enabling multitenancy. The services proposed are application building blocks, from On-the-Shelf pieces of software to virtual machines that have initialized with their Operating System. Yet, this model may not fully describe the current challenges that are facing the business of Cloud-Computing nowadays. The remaining of this section showcases the actual trends in this quickly evolving ecosystem and put more depth to the NIST Definition.

1.2.2 Evolution of Cloud Computing

While the NIST definition [Mell 2011] remains the most widely adopted, State-of-the-Art Cloud technologies have undergone some major changes. Indeed, in the last eight years, we observed a tremendous technology drift: the adoption of containers, the introduction of Functions-as-a-Service, or edge deployments are a few examples of new techniques partially breaking the original definition provided by the NIST. Cloud computing became a solution widely adopted by the industry while it was still a young topic with a wide range of open challenges [El-Gazzar 2016, Senyo 2018]. In this section we will review and analyse some of the evolution that changed the face of the Cloud Computing as it has been originally defined.

1.2.2.1 Adoption of Containers

Another closely related topic to *Cloud Computing* is the containerization. Even if the concept of containerization preceded Cloud Computing by many years, its adoption drastically increased thanks to an Open Source Initiative from 2013 named Docker[1]. Indeed, Docker revealed the potential of containers for Cloud environments; since then containers gradually act as a lightweight substitute to Virtual Machines.

Whereas virtualization creates software-defined components to create a Virtual Machine (VM), containers factor the Linux Kernel by relying on its isolation mechanisms. Containers act as a package embedding a program and its dependencies in a unit having its own life cycle. Both of these techniques create so-called *images*, that are files that can be executed by a dedicated runtime. However, as shown in Figure 1.1, containers share the same and do not embed an Operating System (OS). Therefore, container images are smaller as they do not carry an OS, and their boot time is also reduced compared to VM as the program communicates with a running kernel and does not require to start its own one. However, the abstraction provided by containers is different from the abstraction provided by VMs. The boundaries defined in the service model from the NIST definition do not fit containerization, the definition needs to be extended.

Nowadays, Docker is a widely adopted technology in industry as its benefits are many: it provides the portability for pieces of software along with a distribution and a packaging model while keeping the performance overhead minimal. In this

[1] `https://www.docker.com`: homepage of the Docker project

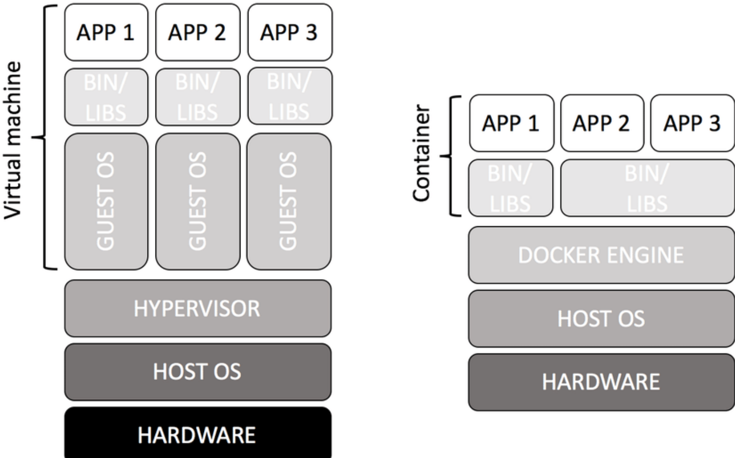

Figure 1.1: Representation of abstraction levels with virtual machines and with containers. Illustration from [Bistarelli 2018]

thesis, we will focus on containerized Cloud Applications, in particular relying on Docker containers. Therefore the next section provides a background study on containerization in order to provide a better description of their performance.

In order to normalize the container ecosystem, after its exponential growth, Docker and other leaders in the container industry (like RedHat, Google or Microsoft to quote few) released in 2015 the *Open Container Initiative (OCI)*[2]. The OCI is an open governance structure modelled after the Linux Foundation whose role is to define the specifications of containers engine. We will be referring to OCI-compliant containers simply as *containers* throughout the rest of this thesis. The notion of containers covers the runtime, the packaging and the distribution models. While docker dominates this market other container engines exists like *Podman*[3], *LXC*[4], or *Youki*[5]. OCI defines the following guidance for each of the following topics related to the container ecosystem:

The isolation and execution runtime: Like VM, containers are files (called *images*) that are executed by a dedicated engine (called a *container runtime*). Instead of creating virtual components (e.g. Hard Drives, Network Interfaces, Memory), the *container runtime* directly uses the host kernel isolation mechanisms, namely *namespaces* and *CGroups* to launch the containerized software. This software does not have the visibility of other processes and resources managed by the host. As a result, when executing a container on a host, a single process is started, the runtime only isolates it from the host. To normalize this model, the OCI defines the *runtime-spec*[6].

[2] `https://opencontainers.org`
[3] `https://podman.io`
[4] `https://linuxcontainers.org`
[5] `https://github.com/containers/youki`
[6] `https://github.com/opencontainers/runtime-spec/blob/v1.0.2/spec.md`: OCI Container runtime specifications

A packaging model: Docker images represent the way containers are packaged. An image can be executed by a container runtime: it embeds the file system containing the program and all its dependencies as well as the configuration required by the runtime to run the program.

A distribution model: The distribution model represents how images are served over the internet and how they are sent over the network.

1.2.2.2 Multiplication of Levels Within the Service Model

Containers have not been the only technology that disrupted the NIST service model; Cloud providers recently introduced the concept of Function-as-a-Service (FaaS). FaaS, also often referred as Serverless, are small pieces of code executed in Cloud infrastructure triggered by external events. They take place in an event-based platform and trigger a function based on an external solicitation, mostly materialized an HTTP Request [Varghese 2018]. FaaS slips between the PaaS and the SaaS layers: it allows running arbitrary code without managing the operational layer, instead of billing for a dedicated platform, the billing corresponds to the number of times the function is used. In the industry, their usage is mainly to offload computations into a data centre to preserve the actual equipment from performing computation intensive workload [van Eyk 2018]. Therefore, by its design, FaaS also enables Edge Computing: developers can decide whether to execute a computation close to the user of offloading intensive ones.

Once again, this new technology establishes a strong boundary between the scope of the Cloud User and the Cloud Provider. In public Clouds, all main actors propose their implementation of the FaaS concept:

- Amazon was the first public Cloud provider to introduce, in 2016, the concept of FaaS within their catalogue with AWS lambdas[7].
- Google followed in 2016 too and also added to their catalogue Cloud Functions[8] which covered a similar scope to AWS lambdas.
- Microsoft also followed in 2016 too and added Azure Functions[9] to their catalogue. Instead of relying on a Linux Base system, Microsoft opted to their own OS as support for the underlying runtimes of languages.

With the multitude of providers supporting FaaS, this raised the question regarding a performance comparison of these services. In [Malawski 2018], the authors created a framework to evaluate the performance of the different FaaS provider. As they hide underlying implementation, the performance evaluation was a black box evaluation. Their framework measured both the time spent to process the whole request (including the time spent on the network) and only the processing time within the function computation. While, in 2018, result showed AWS was having better response time than other implementations, it is to consider that only the time

[7] `https://aws.amazon.com/en/lambda/`
[8] `https://cloud.google.com/functions`
[9] `https://azure.microsoft.com/en-us/services/functions/`

to get the service delivered was considered in the performance evaluation. Indeed CSPs do not allow user-defined functions in FaaS to reach lower-level metrics like CPU usage or memory as it would break the isolation and separation of concerns between Cloud User and Cloud Provider.

1.2.2.3 Deployment Extension

Cloud adoption these last decade has increased at a rapid pace. Nowadays, CSPs host many businesses critical functions for their users, and, therefore, their expectations in terms of performance also increased. In particular, they ask for shorter latency and better availability; CSPs solve these constraints by adding data centre in different locations, closer to the end users. As a result, the Cloud, which has been defined as a centralized model now scatters on multiple data centres around the world. This allows computing resources to be closer to the users, and to replicate data over several "*availability zones*". However, this scattering of computing resources changes the original model of Cloud computing and brings new challenges regarding the placement of services in the different availability zones [Chaczko 2011, Unuvar 2015, Moreno-Vozmediano 2017, Chou 2019].

Notably in [Chou 2019], Facebook published their work where they evaluate the performance of their services deployed at the Edge in order to minimize latency and to better balance the use of their data centre. In this publication, authors modelled edge-to-data centre communication and used this model to maximize the number of user requests the application can handle while optimizing their data centre capacity. With the optimization achieved in their work, they managed to reduce the load of their back-end storage by 17%.

Finally, in [Varghese 2018, Khan 2019], authors review the current technologies as well as their actual challenges to identify research trends regarding Cloud decentralization. In [Varghese 2018], authors focus on the smartphone use case, as it opened a wide range of new opportunities: they have a different usage from traditional computers, involve roaming, access to different data, with more latency constraints. In another hand [Khan 2019] takes a wider scope than only smartphones and considers the IoT and the Big Data use cases where having computing resources closer to data source could improve performance and resource usage. Both of these publications mention *Cloudlets* which are extensions from the data centre positioned closer to the final users, at the "edge" of the network.

1.2.3 Performance Measurements and Concerns

1.2.3.1 From a Provider Perspective

Regardless of the abstraction level or the location of the computing resource, the role of CSP remains to allocate these resources to their tenants. In order to efficiently use their available computing power, most CSPs over-allocate their tenant to their underlying resource. While this leads to a better usage of the overall computing power of the data centre, this over-allocation also leads to a uncertainty for the tenant to really have access to resources it subscribed for.

This over-allocation of resources in data centre remains a well known problem [Yu 2018, Dabbagh 2015b, Dabbagh 2015a], however a good management of this trade-off is critical for CSPs. Also, there is a variety of software implementing computing services allocation to underlying computing resources [Boutin 2014, Verma 2015, Vavilapalli 2013]. While implementations are many, there is no allocation method standing out from the crowd, most implementation are tightly tied to the technology running services. As a result, CSPs leave their allocation methods closed-source and rarely communicate on them, leaving performance studies only black boxes.

In [Yu 2018], authors analysed a 24-hour "trace" dataset from a production in Alibaba data centre, they used this trace to support a study on the Improvement of Resource Utilization (IRU). In this study, a focus is made on the degradation of performance caused by the way a *High Elasticity* co-locates resource intensives services. In this work, authors defines *Plasticity*: the concept Alibaba that used for bounding resources allocated to performance intensives Tasks in Cloud infrastructure. Bounding resources allocated to performance intensive short-lived services allowed them have lesser "noisy-neighboors" issues when co-locating services in their production cluster.

Another work from [Dabbagh 2015a] describes a framework which models Resource Utilization in the context of data centre resource over-commitment. Authors also conducted a study where they analyse the impact of over-committing resources on energy consumption [Dabbagh 2015b]. Both of these works have been based on a "trace" published by Google from one of their +12,000 physical machines cluster. Authors provide an optimization framework that predicts virtual machine allocation to maximize underlying infrastructure utilization and put some machines to sleep to save energy.

Both [Dabbagh 2015a] and [Yu 2018] have been based on real-world scenarios published by notorious companies running production Cloud, respectively Google and Alibaba. Indeed, the impact of over-commitment is almost impossible to model at smaller scale and requires real world data to be analysed.

1.2.3.2 From a Customer Perspective

On the contrary, as a Cloud customer, measuring the performance of a Cloud-deployed application is a completely renewed task: When building an application, developers create an application by composing third-party components with business centric code [Heinrich 2016]. These Cloud resources provided by third-parties *as-a-Service* separate ownership within the Application: maintenance and operations are offloaded to CSPs whereas development and deployment are left to developers. Whereas this abstraction boundary effectively eases deployment, it makes debugging a more complex and tedious task, in particular performance debugging [Jayathilaka 2017].

To assess performance of *services* offered by Cloud Providers, both Customers and providers refer to *QoS* indicators. QoS indicators consists of simple metrics that qualify the performance level of the service provided: they can be measured in Black-Box and do not require any knowledge of the technology executing the service

by the customer. There is a wide number of indicators describing the quality of the service, it varies a lot according to the kind of service measured. In [Singh 2017] a focus is made on the management and on the measurement of the QoS for Cloud resources. Authors provide an extended list based on both experience and a literature review of Performance Indicators used for QoS measurement. While there is a quantity of metrics describing the quality of service applied to Cloud services, the latency/response time measurement outstands from the group and provides a powerful indicator of almost all services offered by a Cloud Provider.

To manage performance, both Cloud provider and customer agree on *Service Level Agreement (SLA)*, an indicator representing a threshold in some QoS metrics.

1.2.4 Closing Words on the Cloud Paradigm

Throughout the years, the concept of Cloud Computing took some distance from the original NIST definition, each of the aspects detailed gained a lot of nuances. First, technologies evolved at an extremely fast pace, Docker adapted the containerization techniques to match Cloud-Computing usage and now, became one of the building blocks of most of cloud technologies. Still, this technology fails to find a representative spot in the NIST service model. Then Edge-Computing changed the perception of Clouds, it brought two main challenges to original Clouds: Location of computing resources and computation offloading. Finally, compared to traditional deployment, Cloud certainly eased designing application but it made them more difficult to monitor as it introduced the concept of *tenant*. Tenants designate Customers and their associated resources. This notion of tenant materialises as an opaque barrier between what is visible by the providers and what is visible by the customer, in term of resources. So when monitoring a Cloud application,wWhereas the application structure has become more complicated, the customers have fewer metrics to monitor to ensure that the application is running under good conditions.

Cloud providers also allowed business specific code to interact with the infrastructure through API. Therefore Applications developers integrated the logic of making an application grow and shrink to follow its usage for example. In the following, we consider how these change on the application infrastructure has affected Software design and therefore the application performance.

1.3 Cloud-Native Application Architecture

The impact of Cloud Computing on application's architecture and software design has been massive: Software systems are now built by selecting, configuring, and composing third-party software-defined services. Also, they are deployed on virtualized, remote infrastructure where the quantity of requested resources can be adjusted at any time [Heinrich 2017]. Therefore, developers building software spend less time creating an infrastructure by using "on-the-shelf-components", so they focus more on the business logic.

Cloud providers build infrastructures capable of addressing the needs of most application developers: the most prominent ones being the almost immediate elasticity

and the virtually infinite scalability of resources. To take benefits of these capabilities, developers have changed their software architecture to adopt a distributed-system structure. The benefits of this structure are many, this allows single components to fail without harming the whole system, therefore allowing to reach the almost 100% availability. Also, it allows granular scalability and deployments; indeed, according to the load of the application, only some components can be scaled to reach the desired performance. Finally, components can be developed and deployed more often and quicker, the organization is not forced to release all their code and features at once, thus making deployments less risky.

The following section discusses the consequences of adopting this paradigm on software engineering. We first discuss the Microservices architecture by presenting its key concepts as well as studies and feedback from companies running large scale microservices application. Then we discuss the evolution of tools commonly used to interface with Cloud Providers and to design distributed applications. This section opens on the challenges of monitoring in this context of fully distributed and resilient application.

1.3.1 The Microservices Architecture

The Microservices approach emerged from industrial running large-scale applications, it consists of an evolution of the Service Oriented Approach (SOA) where each component are materialized by standalone processes communicating with each other over HTTP [Fowler 2014, Newman 2015]. In this thesis, we will not focus ont the strict definition of Microservices, as it is a young topic that still undergoes some adjustments and discussions currently. Indeed some reports from the company Uber, known as a precursor of the Microservices approach, revealed that they already take distance with their early definition [Gluck 2020]. With all this rapidly evolving context, in the following the term “Microservice” will be used to designate a business-centric processes, part of a distributed application, communicating over Application level protocol with other components of the system.

Creating smaller software units contributed to make each piece of a global application simpler. So they are managed by a smaller teams, and capable of a quicker evolution. Application developers take benefits of the *rapid elasticity* of Cloud Providers to scale the number of instances of each service individually as the application needs. Scaling individual components allows the infrastructure to follow the load of the application at a finer grains. Some business components can be more performance expensive than others and only these one are scaled in practice.

In the later, we discuss the nature of this new structure. We emphasize both the distributed aspect and the ephemeral aspect of the multiple services composing an application. Finally we will use studies from real-world applications to back up our words.

1.3.1.1 Modeling Microservices as Service Composition

The way Cloud-Native Applications are built represents a major shift in the way software is developed nowadays. Applications are now divided in multiple in-

dependent components, running on numerous virtual machines hosted into data centres scattered around the world. Separating a monolith into smaller business-centric components is the foundation of the Microservices approach [Fowler 2014, Newman 2015, Gluck 2020]. This approach consists of an evolution of the SOA in which components consists of standalone processes. These processes communicate with each other over the network through API calls (also often designated as Remote Procedure Calls). The Netflix company that adopted Microservices to structure its main application referred this approach as *fine-grained SOA* [Heinrich 2017]. In the recent years, developers adopted this approach that borrowed many concepts from distributed computing and network engineering. Application can now be modelled as distributed systems, they are designed to be decentralized, to avoid Single Points of Failures, to have self-healing capabilities.

In [Sampaio 2019], authors present an autonomic approach based on the MAPE-K control loop. The MAPE-K control loop is a structure defining a self-managed computing system. This control-loop is fed by monitoring data, which is then analysed in order to identify actions the system has to take on itself in order to be self-maintained. This control loop is supported by a model, rules and properties maintained by a knowledge-basis at runtime [IBM 2005]. In this work, authors use a state-of-the-art application and tools. Too feed their control-loop, authors pick traditional monitoring sources from each individual components (some metrics, logs, and traces). The goal of the control loop was to re-allocate microservices in order to find their best placement onto the infrastructure. Their work demonstrated that placement achieved by current tools are not efficient and that a lot of resources can be saved up on host by finding better placement for microservices. Their work highlighted also the complexity of the problem of finding the optimal placement for a given application. Finally, authors reported their difficulty with current tooling to get relevant monitoring data at an acceptable performance cost.

SOA architecture has been looked from many different angles in literature, and studies have taken many different approaches regarding optimizing an SOA application. In these papers [Alrifai 2009, Alrifai 2010], authors provide a method that optimizes the composition of web services under QoS constraints. The main interesting aspect of this work is that it tackles the performance problem from a runtime selection point of view instead of taking part of resource placement perspective. Opting for a runtime selection of services in a SOA application is identical of a load-balancing problem in the context of a Cloud internal network of microservices.

1.3.1.2 A Rise of Ephemeral Resources and Over-Allocation Issues

To keep each microservice under a manageable size, they are often associated with a single business function; while some functions are designed to stay for a long period of time (like serving user requests), others are designed to be short-lived functions. In practice, this takes the form of separated computing resources in Cloud Applications, Business functions designated to stay and deliver a service are often referred as “Services” whereas short-lived function are referred as “Jobs”. These jobs may consist of continuous validation of data or data management and are especially encountered in the case of stateful services. These ephemeral jobs are

often computation intensive, and, to preserve other service performances, they are scheduled in dedicated virtual resources.

Reports from large scale applications demonstrate the importance and the performance impact of these functions. In [Ardelean 2018], Google research department published a performance study of their GMail application: these functions are designated by "*non-UVR*", which stands for non User Visible Requests. In this publication, non-UVRs cover all action generated by the application on itself: examples provided are either data management actions and maintenance such as repairs and software updates. Throughout this performance study non-UVR contribute to a great amount of the global application load. While it is not the topic of this work in particular, this publication also illustrate the scheduling challenges of these short lived function.

In [Yu 2018], authors studies this allocation from real world perspective; they use data published by the Alibaba company to analyse their scheduling techniques and technology for short-lived function while the main application kept on delivering the service under acceptable objectives. Results of the study highlight a custom technique used by Alibaba engineers to limit the elasticity in order to leave enougth computing power to other computation intensive services in their Cloud Application. Indeed, considering the common practice in virtualization of over-allocating resources, the execution of computation-intensive services may have side effects which cause uncertainty on the Quality of Service of others.

Even if this over-allocation problem and its side effects are well studied and has been tackled down from different angles, there is still no consensus on the trading between resource usage and the guarantee of respecting the expected SLA.Over-allocation still is a critical issue that may only be observed in real-world scenarios and that remains a business-centric challenge for companies. As a result, academic publications, instead of being ahead of time, take the form of feedback from the industry, often hiding some important data for the sake of confidentiality. In addition of the Google example provided earlier, the company Facebook made numerous publications in the domain of data centre management [Boutin 2014, Veeraraghavan 2016, Kaldor 2017, Veeraraghavan 2018, Chou 2019]. These publications covers the traffic engineering and migration techniques the company resorted at the scale of the globally deployed Facebook Application. Although, in [Grandi 2016], Microsoft published a study on improving scheduling policies for jobs in a production analytic cluster. While the final study on new policies was run on a simulation target, original data and problem have been observed from actual monitoring data gathered from production cluster. The monitoring data considered for the simulation to represent the cloud environment is the same trace that Google made public that has already been mentioned in Section 1.2.3.1.

1.3.1.3 State-of-the-Art from Large Scale Cloud Applications

As we observed the expansion of Cloud Providers like AWS in the recent years, we have also witnessed the apparition of many new web based Applications fully exploiting their capabilities in term of resource pooling: Netflix, Uber, Twitter, Twitch, AirBnB, Zoom are a few examples of AWS-powered applications.

Some of these applications took gigantic proportions, Uber being one of them, and they had to engineer quickly an application capable of using Cloud scaling capabilities. Engineers made several communications through corporate channels, to explain and detail the challenges they faced. In a blog post [Haddad 2018], author provides a feedback on their implementation of Microservices; five years later, in [Gluck 2020], they publish an update on their method to design global applications. In the later, author mention that Uber is made of around 2,200 critical microservices, and, when considering non critical ones the order of magnitudes climbs [Gud 2019]. Managing such an amount of microservices causes scaling challenges as there are no monitoring tools capable of proving a unified and wider view of a distributed Cloud application. Other companies went talky about their internal structure and the way they handled distributed software design in engineering conferences. A lot of them published their so-called "Deathstars": a global map of the communications between microservices. Figure 1.2 provides maps from the Amazon and Netflix apps gathered from Engineering conferences [Cockcroft 2016a, Cockcroft 2016b].

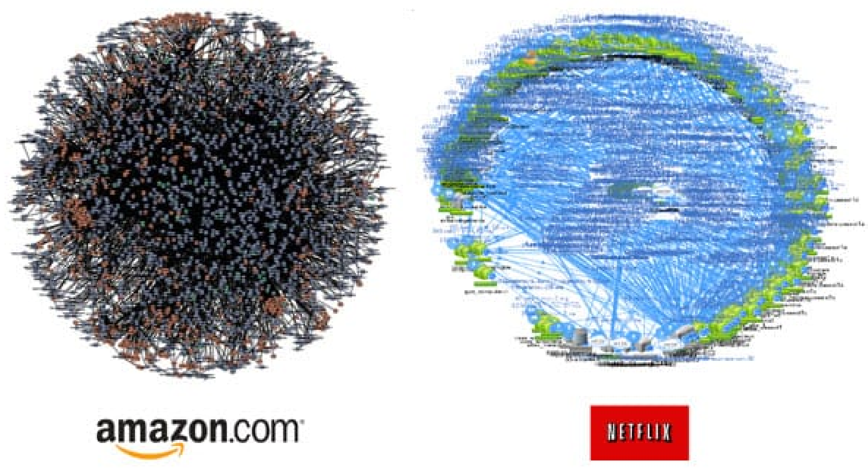

Figure 1.2: Amazon and Netflix "Death Stars" from 2016

Throughout their evolution, these companies frequently reported their difficulties scaling their application through academic conferences, white papers and/or technical blogs. A particular focus is often made on scaling monitoring tools: indeed in a distributed context the number of processes, or virtual machine to monitor is rapidly evolving. In 2013, Quantacast reported to monitor around more than 2,000,000 unique metrics [Quantcast 2013]. In 2014, Netflix also reported to monitor the same number of metrics to ensure the health of their application [Netflix 2014]. Finally, in 2016, Uber reported monitoring more than 500,000,000 unique series to monitor their global application [Uber 2016]. All these companies reported struggling scaling their monitoring to the size of their application, throughout their massive data gathering.

1.3.2 The Evolution of Cloud Resource Management

One of the major advances provided by Cloud technologies has been the capability of controlling the resources with business specific code. Indeed, all CSPs expose an API that allows to request or release virtual resources without any human interaction. This capability has been heavily used as a medium to execute feedback from monitoring alerts. The main example being scaling up and down instances of a service based on its CPU and memory usage. As a result, in Cloud-Native Applications, interactions with the CSP took a prominent place in their design: infrastructure became dynamic and the strong boundary between the infrastructure and the business specific code blurred.

With this dynamic infrastructure we saw the creation of Cloud Specific software that manage life cycles of each of the components of the Application. We first witnessed the adoption of Infrastructure as Code (IaC) where developers define through a Domain Specific Language (DSL) the resources used by their applications. Examples are *Puppet*, *Ansible*, *Terraform*. These applications manage the life cycle of resources defined in the manifests written by developers, ensuring migrations of the state held by services. Then, more elaborated tools appeared where this logic was directly integrated into the Application itself through a **control plane**. Unlike, IaC which computes changes only when developers trigger a new manifest, control planes apply changes on the infrastructure at runtime. The most famous control plane software is Kubernetes, it allows managing containers across multiple machines.

1.3.2.1 Container Lifecycle Management with Kubernetes

Kubernetes is one of the most popular Open-Source Container Orchestrators; it has been initially open sourced by Google and is now hosted by the Cloud Native Computing Foundation (CNCF). This project is the Open source version of *Borg* [Verma 2015], which is the technology that Google used internally to deploy their applications. Kubernetes is today considered as the State-of-the-Art way of deploying a containerized application at scale. Today, many CSP provide managed Kubernetes services.

Kubernetes' role is to schedule and follow the execution of containers under a multitude of machines. It acts as a DSL describing the distributions of workloads under multiple executors; this DSL is understood by a control plane that manages the life cycle of each individual workload in the system. Like other orchestrators, Kubernetes is capable of detecting some issues within the system and can take some countermeasures; although scenarios managed by the orchestrator are rather minimal:

1. **Replication Controllers** allow to automatically restart microservices when failures are detected.

2. **Horizontal Pods Auto-Scalers** allows the number of instances of a particular microservice to be scaled up / down based on the amount of resources the container uses (CPU cycles or memory consumption).

3. Most cloud providers deploy Kubernetes clusters based on an **Automatically Scaling Group** which makes the number of machines executing containers be dynamically adjusted based on the amount of resources requested when scheduling containers on the cluster.

As a result, some basic scenarios for handling software bugs and hardware failures are automatically covered by the control plane, therefore minimizing their impact in production. Other recent initiatives decided to push the limits of Kubernetes capabilities by extending it to manage with greater granularity the interactions between components. This is done by embedding HTTP proxies in each microservices where their configuration is managed by a logically centralized but physically distributed control plane.

1.3.2.2 Instrumenting the Cloud Application Network

There were also other benefits brought by this new paradigm: Uber detailed some techniques that have been used to test code in production thanks to micro services structure and some internal traffic routing [Gud 2019]:

1. **A/B Testing** corresponds to routing traffic either to an instance of a given service or to another having minor differences. It allows to measure the impact of these minor differences on the global application performance, or the user adoption of these new changes. For example, for a given microservice with two different instances where A is experimental and B is production; a portion of the traffic normally targeted to B is sent to A instead.

2. **Blue/Green Deployments** corresponds to gradually shifting traffic received by a service from a version called *Blue* to a version called *Green*. Instances of *Blue* microservices are gradually drained whereas instances of the *Green* version are gradually provisioned. It allows application developers to better manage the deployment of new version of each component while ensuring the service remains online and available.

3. **Canary Deployment** corresponds to shadowing a portion of the traffic of a production microservice to a testing instance. While the testing instance is not involved in the response returned to the client, it allows application developers to test in the real case scenario the behaviour of a specific microservice.

These scenarios are not made possible with standards Kubernetes implementations and require either to add specific code in each component to handle request routing either a particular instrumentation to apply the network configuration finely on each individual component. To keep microservices simple and to segregate network logic from business code, this role is often handled by Service-Meshes that inject individual HTTP proxies that are managed by a centralized control plane [Li 2019].

1.3.3 Challenges Regarding Performance Evaluation

1.3.3.1 Distributed Systems Monitoring Challenges

Therefore, developers embraced the "distributed system" design for Cloud-Native Application. The benefits have been numerous: the distributed structure brought an almost infinite scalability for critical components. Also, segregating resources brought more granularity on the management of component lifecycles. Still, the distributed architecture also came with its drawbacks. In [Kendall 1994], authors explain that the abstraction efforts made to represent a distributed application as a standalone and coherent entity have a negative effect on the monitoring of the platform. They hide most of the failures that can occur in a distributed application.

> The hard problems in distributed computing are not the problems of how to get things on and off the wire. The hard problems in distributed computing concern dealing with partial failure and the lack of a central resource manager. The hard problems in distributed computing concern insuring adequate performance and dealing with problems of concurrency. The hard problems have to do with differences in memory access paradigms between local and distributed entities. ([Kendall 1994])

In another work [Woodruff 2017], authors provide an extended study of fundamental problems behind distributed computation: they study complexity of algorithm commonly used in the message passing model. In this paper, authors prove that computing exact solutions to many basic statistical and graph problems used to model the message-passing paradigm are communication inefficient. There are two consequences: in real world scenarios, we often observe a relaxation of the problem and approximations to lessen the overall complexity of the problem. In addition, communications contribute to a greater amount of time of the global latency of the system. Computation cannot be both communication efficient and resource efficient while bringing exact results.

1.4 Cloud Application Monitoring

As developers structure their applications as distributed systems, performance analysis and debugging tools also need to adapt these changes in design. Monitoring and troubleshooting distributed applications is notoriously a hard problem. Beside detecting software and hardware failures, Cloud monitoring tools must also be proficient at addressing distributed system issues, notably misconfigurations, noisy neighbours or bottlenecks.

1.4.1 Cloud Application Monitoring and Performance Analysis

1.4.1.1 Cloud Application Monitoring: A Scaling Challenge

With the exponential growth of the number of systems to manage, monitoring a Cloud Native Application has become a tedious task. Indeed, the vast majority

of monitoring tools were inherited from traditional standalone system monitoring. These monitoring tools usually collect two types of data:

Metrics They associate numerical values with timestamps, they are often qualified as time series. They can represent, for example, CPU usage of a virtual machine, the amount of memory used by a program, the number of items stored by a message queue.

Logs They associate textual values with timestamps, this textual value can be structured (e.g. JSON, protobuf or any serialization format). They represent events within a system like, for example, a user's request on the proxy, an authentication attempt on the application, the decision by a component to ask for more resources to the CSP, …

Multiple big tech companies reported that traditional monitoring does not scale well considering the number of components involved in a Cloud Application. Each application now being divided into multiple standalone system the amount of time series or logging source has tremendously increased.

Indeed, in 2014, both Quantacast and Netflix reported using 2,000,000 unique series for monitoring their Cloud Services; the same year Uber reported monitoring 500,000,000 unique series [Thalheim 2017, Quantcast 2013]. This high number of metrics results in a higher monitoring cost, as it now requires machine learning and big data techniques to identify performance anomalies [Dalmazo 2017, Dean 2014, Gan 2018a, Gan 2020, Nedelkoski 2019]. However, the scope of monitoring is wider: all of the quoted works used computation expensive techniques to establish causality between measurements, or between heterogeneous data sources characterizing the same system.

Logging has also undergone some massive scaling in the context of Cloud Applications: in [Chow 2014], Facebook provides feedback on using logging messages to investigate latency issues. Throughout their work, authors used over 1.3 million HTTP logs to study causality among component of their distributed application. This augmentation increases complexity to extract significant signals from this amount of data. While they managed to discover the structure of their application only by analysing logs, the processing power required to extract dependency information took the form of a powerful Spark Cluster running for hours. This work marked the starting point of Facebook studies on the dependencies untangling of their whole application. The following sections provide a comprehensive literature study of their work regarding optimization of their massive Cloud Application.

1.4.1.2 The Case of the Facebook Application

At the scale of a globally used application, made of hundreds of services geographically distributed on multiple data centres, optimizing performance involves minimizing network latency while keeping the utilization of data centres as low as possible. Facebook Engineering published various papers where they detail how they used Traffic Engineering to preserve the balance between latency and data centre utilization.

In [Veeraraghavan 2016], authors detail a solution that manages the traffic generated by users into a geographically distributed application. Their custom traffic management improves hardware usage in production by 20%. In [Kumar 2018], authors raised the problem of congestion and bottleneck links in Cloud-Application. They address these problems with the implementation of a routing algorithm dedicated to service-to-service communications, balancing network calls in a more efficient way. In addition, *Maelstrom* [Veeraraghavan 2018] applies traffic engineering techniques to disaster mitigation and recovery. Finally, in [Chou 2019], authors present *Taiji*, an application-level load balancer that pushes more in depth the routing and placement of computing resources to the edge. In this contribution, adding application-level parameters to the routing algorithm reduces the load of back-end servers by 17%.

These various publications made on a real large-scale application (Facebook) highlight that the importance of a smarter in-app request routing can greatly improve a resource management in a multi data centre cloud. However, methods used in these publications are not generic and result of years of engineering based on the specificity of the Facebook application.

1.4.1.3 Untangling Services Dependencies in a Cloud Environment

Facebook Engineering team has not been the only team to study dependencies appearing in their system. This field is an active area of research that has been catching the interest of both academics and companies over the recent years.

A study from Google on the performance analysis of the GMail application has highlighted that resource usage of communicating processes are not independent from each other [Ardelean 2018]. In this publication Google expresses the balance of the load generated by the user and also by its ephemeral tasks. To precisely measure performance on a multitude of machines on scattered on multiple data centres the paper details a technique where engineers directly modified the kernel of the hosts: through the use of a particular system call they could sync performance capture on multiple data centres, allowing capturing a trace embedding data with precision. The data captured in the trace crossed multiple abstraction boundaries, and the manipulation is feasible by Google as they manage the whole Cloud, from the physical machine to the application.

In [Lin 2018], authors also instrument the kernel of the machines running a Cloud Application. However, this work focuses on Kubernetes Clusters. With a Custom Kernel instrumentation, authors have been able to extract the information held in network calls so that their system keep track of which services communicates to which service within the cluster. In this work, authors were able to maintain causality graphs of the services deployed on a Kubernetes Cluster. This root-cause analysis is performed in the context of Service Level Objective (SLO) monitoring to help identify the source of the latency induced by each service.

In [Jayathilaka 2017], authors leverage HTTP headers to inject a unique ID for each incoming request to allow them to observe the propagation of a single request within the whole application. Dependencies are observed at runtime through HTTP logs and authors rely on this dependency graph to perform root cause analysis. The

implementation of this work is specific to a PaaS platform. However the method of generating a unique ID for each incoming request is now well known as it is the state-of-the-art way to build a trace [Kanzhelev 2020].

As a result, implementations are many, and most of the time context specific to a company [Ardelean 2018, Kumar 2018]. Most cloud monitoring techniques observed in this section used to break the software isolation layers, and have been done by companies owning their data centre. The works presented in this section stay implementation specific and only available for the owner of the Cloud infrastructure. The problems tackled, however, remains at the application level. In the next section, we focus on the normalization initiatives that have been observed around the Cloud ecosystem that brings this visibility by respecting software isolation properties.

1.4.2 Toward a Unified Cloud Application Monitoring Framework

Whereas monitoring tools tend to become more and more exhaustive in the way they describe Cloud environments, cloud orchestrators and providers, on the contrary, tend to obscure underlying implementations, making debugging more difficult [Jayathilaka 2017]. The recent *OpenTelemetry* initiative aims to normalize Cloud monitoring by providing an open format and production-ready binaries for Cloud-Native Monitoring. While the project is still in Beta at the time of writing, it opens many research opportunities to enhance the quality of tracing data or the way it is processed.

OpenTelemetry covers a wide range of issues related to Cloud monitoring, in particular it defines a semantic that would be shared all the entities of the Cloud to provide standardized monitoring data. In addition, OpenTelemetry defines a way of establishing correlation between measurements in a distributed context. This technique is called context propagation and is supervised by the W3C [Kanzhelev 2020]. Context propagation enabled the Distributed Tracing by propagating along with application data a unique ID that identifies the original action that triggered the complete system. With context propagation, it is now possible to reconstruct execution traces of a distributed application. *OpenTelemetry* aims to provide a normalized format for traces as well as other monitoring data sources like metrics and logs. And, while this section does not dive in the technical implementation of the tool, section 2.2 will provide a more detailed introduction to the technology.

The project is not considered mature yet, but it inherits from its predecessors *OpenTracing* and *OpenCensus* which were complementary projects already having the goal of normalizing cloud monitoring. There are also works closely related to *OpenTelemetry*, in particular for the purpose of enhancing context propagation with additional contextual information and to create more complex graph of function calls [Mace 2018a, Mace 2018b]. Distributed Tracing also become an increasingly adopted monitoring source in the industry, in [Las-Casas 2018] and in [Las-Casas 2019] authors provide a study that details sparse sampling methods for distributed traces.

There are also ongoing works that leverage traces as a new monitoring data source to work on anomaly detection [Nedelkoski 2019] or on a visualization technique for debugging distributed systems [Anand 2020]. Even if traces are still rarely

used, the heavily connected nature of tracing data could constitute an opportunity to tackle down some Cloud-specific monitoring challenges.

1.4.3 Research Challenges for Monitoring Cloud Application

In recent research works, a lot of attention has been brought to the impact of these emerging constraints related to monitoring heavily distributed applications. Numerous academic publications address the challenges of Cloud Native performance evaluation and monitoring. In particular, in [Heinrich 2017], authors raise the research challenges linked to an extremely dynamic environment: development practices commonly adopted in the industry like Continuous Integration / Continuous Deployment (CI/CD) allows performing tests on a subset of the application. While they acknowledge the use of these practices, they also raise the additional difficulties and challenges of monitoring the software and modelling its performance. The cloud ecosystem can be considered to be a volatile environment made of a heterogeneous multiplication of technologies.

Still, this view does not cover all the challenges reported by the technological companies to this date. The next sections focus on cloud application specific scenarios reported to have a great impact on improving a Cloud application performance.

1.4.3.1 The Multiplications of Tools and a Lack of Global Vision of the Application

In [Da Cunha Rodrigues 2016], authors address the actual complexity of monitoring tools in the domain of Cloud applications. With the multiplication of entities to monitor and all the abstraction layers proposed by CSPs, the number of dedicated tools involved in monitoring an application is getting considerably high. And, while authors remain on the strict definition of the abstractions levels provided by the NIST (IaaS, PaaS and SaaS), the number of tools used in performance evaluation and analysis still is too high to go toward a unified view of Cloud performance evaluation.

Further, these tools do not collaborate with each other and keep monitoring data under their own “realm”. And, while it is normal for an application to compose services from different abstraction layers, these monitoring tools do not share data or context. Therefore, this makes establishing a correlation among the partial vision held by each of these tools a complex and open problem. The *OpenTelemetry* initiative aims to provide a format allowing the multiple data sources (Logs, Metrics and Traces) to share some context and establish a correlation between these data, all managed by the tooling. Still, while this framework is promising and supported by companies, it is still in beta and is not widely adopted yet. With *OpenTelemetry*, establishing correlation between multiple data sources from within the tools would allow to group and form clusters of different type of monitoring data (logs, metrics and traces) around the concepts of observed event. While the project is too young to have concrete examples of such aggregations yet, actual implementation and standardization heavily rely on the concept of context propagation to enrich monitoring data with meaningfull context.

Finally, in [Heger 2017], authors focus on a higher-level view of monitoring a Cloud application instead of focusing only on its individual components. Notably, authors focus on the usage and interpretation monitoring data. They extend the scope of problem detection and alerting for each individual component to new topics: mainly root-cause analysis and problem diagnosis which are two challenges of distributed system monitoring. The next section provides a study on how bottleneck identification and Root Cause Analysis are tackled down in Cloud Computing environment by current research works.

1.4.3.2 Root Cause Analysis, Bottleneck and Chokepoint Identification

Bottleneck and chokepoint identification in a complex environment still take a prominent place in actual literature. In [Marvasti 2013], the company *VMWare*, which dominates the market of virtualization technologies, presents a model for evaluating the performance of virtualized IT environments. A particular attention is brought on identifying bottlenecks in complex systems made of a multitude of subsystems. While representing company IT infrastructure, the monitoring problem addressed remains valid in distributed applications. In this paper, authors use a recommendation system to associate a new diagnosis to an aggregation of automatically detected failures of a system. The data used to support the recommendation engine is the history of events already classified by human operators on similar infrastructure. In this work, bottlenecks are detected by selecting the resources presenting a higher presence of networking failures, in particular delays.

Other publications studied bottleneck identification under a different angle: for example, in [Gan 2018a], authors present a performance model that uses queuing networks to identify the propagation of a bottleneck in a system. However, their definition of bottleneck differs from the previous work, while it was centred on networking errors and delays.

Also, in [Ibidunmoye 2015] authors provide an extensive literature study of the topic of bottleneck and chokepoint identification. This topic is not only applied to Cloud application monitoring but extends to a wide variety of distributed systems. Nevertheless, authors address different kinds of bottlenecks that complete the vision brought by previous studies : In a multi-tenant environment, noisy neighbours can have a great impact on an application performance. Indeed, CSPs often overcommit computing and network resources. Therefore a tenant can have its underlying resources drained by another performance-hungry tenant.

This literature review samples some performance studies in Cloud environment addressing bottleneck identification but each with a particular definition of a bottleneck in a Cloud environment. All these definitions have their use cases and demonstrate that bottleneck identification is still an active research topic. Also, in [Ardelean 2018], Google engineers provide feedback on the performance evaluation of the GMail application. While this study is not plainly focused on Bottleneck Identification, it uses common mechanisms and problems: Monitoring a massive Cloud Application like GMail involved more network monitoring. In a distributed system of this scale, authors reported their main challenge was to have an automatic root-cause analysis mechanism.

In [Jayathilaka 2017], authors instrument an open source PaaS platform to establish causality of events and perform Root-Causes analysis. Their implementation remains technically very close to *OpenTelemetry* context propagation and enabled authors to perform Root-Cause Analysis at runtime. While *OpenTelemetry* was not mentioned in the paper, tagging user request with an ID and analysing the propagation of this ID in the system is the technique that enabled distributed tracing. Therefore, this work demonstrates some potential use cases for this new monitoring format to perform Root-Cause Analysis in a Cloud Application. Also, in [Lin 2018], authors propose a graph based technique that instruments a Kubernetes Cluster to perform similar Root-Cause Analysis scenarios. This work also relies on a custom instrumentation technique used by the authors to build a causality graph. This implementation also observes network communication within a Cloud Application to establish the Causal relationships between components.

Therefore, analysing bottlenecks and identifying the Root-Cause still remain an active research topic open to various interpretations. Through the *OpenTelemetry* initiative, networking observability in a Cloud become possible in a standardized way.

1.5 Scope of the Thesis

Over this chapter we presented multiple concepts and research challenges related to Cloud computing and Cloud application performance monitoring. The domain covered is vast and there are multiple valid perspectives to tackle down the problem of Cloud Application performance analysis. The work presented in this thesis echoes the development of an application designed at Orange until 2020: *Djingo*. Indeed, in its original design, *Djingo* was a voice assistant embedded in a speaker device. This device used to communicate with a Cloud hosted application capable of processing the audio signal to extracts the words, decode the intent of the user request and then route to the service capable of addressing user requests.

From an architectural point of view the application was developed with the micro-service approach, scattered in multiple data centre, called third-party services, and was instrumented to emit traces. The application was instrumented with State-of-the-Art cloud monitoring tools and faced performance issues during its development. Whereas each component was individually monitored through metric gathering and log collections, the lack of global vision of the application was armful and some performance issues required manual human investigations. The most common question was detecting the cause of the latency observed, in particular when the SLA were not respected.

In this thesis we take the opportunity of having a large-scale application with this new monitoring data being traces. Indeed, the driving idea presented in this document is to use the highly connected data presented in traces to investigate performances anomalies where State-of-the-Art tools have been lacklustre. Therefore, while this document does not invalidate the value of logging and metrics for performance monitoring, the challenges presented here aims to use tracing not only for debugging purpose but for production performance investigations. Indeed, current

cloud monitoring still fail to provide a high-level view and exhibit issues in component interactions. Traces, as a monitoring data source, offers the opportunity to exhibit such behaviours and *Djingo* has been a motivating example to be as close as possible to real-world performance issues.

Throughout the rest of this thesis, we propose to address performance issues and challenges that have been raised by both developers and operational teams. In particular, it was complex to observe and qualify the path taken by an incoming request in the whole system; a particular concern was that some request may have higher latency due to inefficient routing among all the micro-services of the application. Also, another concern of the team was the identification of bottlenecks at runtime, especially during the load tests of the application. And, while each service was monitored individually, the system did not help for root-cause analysis and fell short for explainability of performance results.

The work presented here leverages the uses of traces and, in particular of normalized tracing to exhibit inefficient service composition and hotpoints identification in a vast system. Still, Djingo development was stopped a bit after the COVID-19 lockdown which led to difficulties for evaluating the approaches taken. Nevertheless, whereas the following experimentation part have been run on smaller-sized projects or under a simulation environment, the underlying questions come from the development of a large-scale application with heavy performance requirements.

1.6 Conclusions

In this chapter we reviewed the research literature on Cloud Computing in general. A particular focus has been made on the performance evaluation of Cloud-based Applications as well as their performance issues. We observed that, throughout the years, the definition of Cloud has become to be more and more complex: many tools and techniques which now take a prominent part in Cloud software design fall out of the scope of the traditional NIST definition which still acts as a reference nowadays. This initial definition can be enriched by new terms and concepts like concept of containers, Functions-as-a-Service or the multi-data centre deployments.

Software performance evaluation has also undergone massive changes: Cloud applications are now distributed system which can have different types of failures, not covered by traditional standalone system monitoring. When perceived as a distributed system, the network linking Cloud components take a prominent role within an application performance. Literature tackles down these challenges imposed by the distributed nature of Cloud Applications by making customisation in the network stack.

Also, under the vision of the CNCF, an open source project named *OpenTelemetry* proposes to standardize tracing of Cloud Applications. While this project is extremely young, it proposes to normalize Cloud Application Monitoring by defining a common format for all the types of monitoring data: a common format for metrics, logs and now traces. Traces are the only format capable of expressing the causality of events, and many companies and researchers reported using tracing system to solve performance issues related to the distributed nature of Cloud Applications.

Chapter 2

Modelling Cloud Application Structure with Hierarchical Property Graphs

Contents

2.1 Introduction **33**

2.2 Distributed Tracing Ecosystem **35**

2.2.1 OpenTelemetry Data Collection Architecture 35

2.2.2 Collecting Traces in Jaeger Tracing 37

2.2.3 The Jaeger Analytic Library and its Limitations 38

2.3 Extracting a Structural Model from Traces **39**

2.3.1 Identifying Common Elements to Aggregate Traces 39

2.3.2 Modelling an Application Hierarchical Structure 41

2.4 Modelling Components Interactions **43**

2.4.1 Leveraging the Property Graph Model to Identify the Type of Communication 43

2.4.2 Graph Rewriting Operations 45

2.4.3 Building a Hierarchical Property Graph 49

2.5 Implementation **50**

2.5.1 Extracting Data from a Jaeger gRPC Endpoint 51

2.5.2 Property Graph Encoding 52

2.5.3 Graph Rewriting Operations 55

2.6 Conclusions **57**

2.1 Introduction

In the previous chapter, we provided a context for Cloud-Native application monitoring. We identified OpenTelemetry traces as a promising data source that expresses relationships among performance measurements in a distributed context. Also, while research initiatives on untangling dependencies in a Cloud environment have been many, there are still few that reached the “production” stage within the industry. Obtaining a wider view of a distributed application is still an active research topic. Over the past years, distributing tracing has become more considered

by both researchers and software engineers to observe components interactions in a Cloud Application.

Distributed tracing provides means, through instrumentation and with a minimal overhead, of formalizing dependencies between entities taking part in a Cloud Application. And, while implementations have been many, the OpenTelemetry initiative helped Cloud-Native Application Monitoring to make a step toward standardization. Indeed, behind the designation OpenTelemetry lies multiple concepts and pieces of software: the **data semantic**, the **software instrumentation libraries** for specific languages, the **collectors processes** that form a pipeline and, finally, the **storage backends** making data available for further processing. Whereas the initiative covers in detail all these topics, we will mainly focus on the two ends of the pipeline: the data semantic normalizing telemetry data and the storage backend.

So, in this chapter, we propose a model leveraging the complete OpenTelemetry ecosystem. This model builds and maintains, at runtime, an application-wide view of its performance by using traces. It materializes as a hierarchical property graph whose vertices are entities executing some services, relationships of the graph that demonstrates some dependencies. These dependencies can take two forms: *structural dependencies* and *flow dependencies* which translates into either an entity is executed onto another either the entity requests another one. It uses OpenTelemetry resource semantic to represent resources as vertices and the relationships expressed in traces to create the flow dependencies. Indeed, when considering traces, they appear to be more than a collection of data objects living in a multi-dimensional space independently. The core of tracing data resides in the interdependencies expressed between measurements. Property graphs provide a powerful machinery to represent these traces; the capability to have labels and attributes on both vertices and edges allows the preservation of the original data semantic.

Throughout this chapter, we provide a study focussed on the distributed tracing technology. We propose the model that leverages the data semantic and the traces to build and structure a hierarchical description of the application. The study is organized as follows:

Section 2.2 introduces the key elements of the OpenTelemetry ecosystem, from the data collection architecture to its actual data model for traces.

Section 2.3 presents a way to use both data semantic and relationships expressed in traces to encode a property graph describing the hierarchical structure of an application.

Section 2.4 describes the graph rewriting approach that has been developed to highlight the different levels of communications involved in a geographically distributed context. It also formalizes graph rewriting with graph pattern matching sequences to provide a generalized approach of our model.

Section 2.5 describes the implementation used in this thesis that process an online flow of traces to maintain, at runtime a hierarchical model in a graph database.

2.2 Distributed Tracing Ecosystem

2.2.1 OpenTelemetry Data Collection Architecture

The OpenTelemetry initiative covers a wide scope: standardizing Cloud Application monitoring to allow multiple tools and vendors to interconnect. And, while we focus on the tracing specifications in the scope of this thesis, the project also includes semantic definitions for metrics and structured logs.

Whereas monitoring solutions applied to Cloud Computing have been myriad over the past years, OpenTelemetry stood out from the crowd by being a standard for creating observability data and not being an Observability platform. Observability platforms are commercial software that process monitoring data to extract alerts, examples are : *Opsani*[1], *New Relic*[2], *Lightstep*[3] and *Datadog*[4] to quote few.

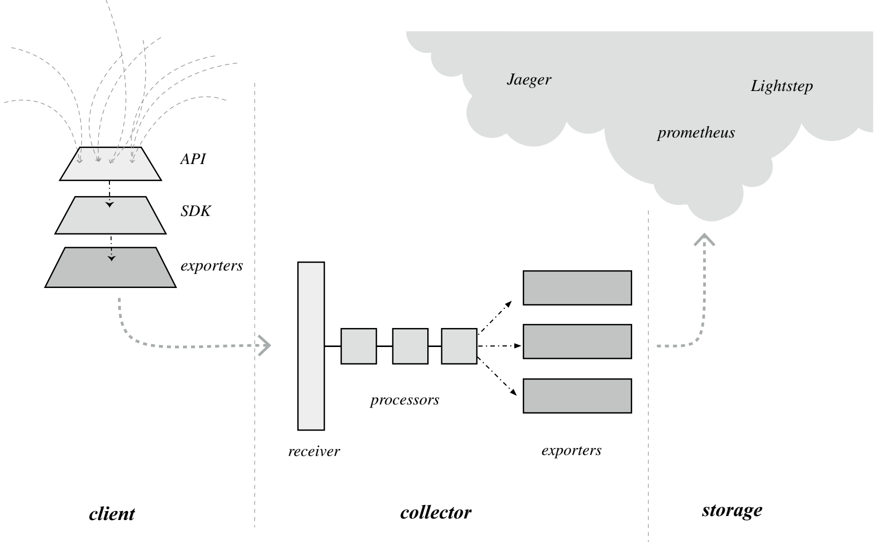

Figure 2.1: OpenTelemetry Pipeline Architecture

Figure 2.1 describes OpenTelemetry components; they take the form of a data pipeline collecting monitoring data from instrumented application. This pipeline is made of three stages:

The Client collects and formats monitoring data to match the semantic defined in the *standard*. Clients take the form of libraries included into the code or standalone processes collocated with the monitored microservice.

The Collector receives monitoring data from multiple clients, this data is then processed to be sent to other collectors or to storage applications. Processing done in collectors includes enriching monitoring with metadata, sparse sampling or filters.

[1] `https://opsani.com`
[2] `https://newrelic.com`
[3] `https://lightstep.com`
[4] `https://www.datadoghq.com`

The Storage is the ultimate step of the pipeline, once monitoring data has all attributes set, it is delivered to storage backends dedicated to exploit this data. These storage applications may either be simple storage/visualization application like *Prometheus*[5] and *Jaeger Tracing*[6] or Observability platforms.

OpenTelemetry pipeline has been designed to be able to chain multiple collectors one after another. Therefore, the pipeline can match the expectations of a physically distributed application by providing multiple points collecting data in each data centre. For example, collectors can have local authority on the microservices hosted in their data centre, providing additional context that follows the semantic, for then referring to a central cloud collector that aggregates monitoring data from multiple locations. Table 2.1 provides a sample of the semantic characterizing the location of a microservice within a Cloud platform.

Attribute	Description	Examples
`cloud.provider`	Name of the cloud provider.	`alibaba_cloud`
`cloud.account.id`	The cloud account ID the resource is assigned to.	`111111111111`; `opentelemetry`
`cloud.region`	The geographical region the resource is running. Refer to your provider's docs to see the available regions, for example Alibaba Cloud regions, AWS regions, Azure regions, or Google Cloud regions.	`us-central1`; `us-east-1`
`cloud.availability_zone`	Cloud regions often have multiple, isolated locations known as zones to increase availability. Availability zone represents the zone where the resource is running.	`us-east-1c`
`cloud.platform`	The cloud platform in use.	`alibaba_cloud_ecs`

Table 2.1: OpenTelemetry Cloud Semantic for Resource Location

[5] `https://prometheus.io` An open-source system monitoring and alerting toolkit build around a time series database discovering services and collecting metrics.

[6] `https://www.jaegertracing.io` An open-source distributed tracing system that helps provide insight of microservices application and distributed systems.

The project has the capabilities of describing and monitoring a complex environment made of multiple layers of resource locations involving a geographical distribution of data centres. This is an opportunity for Application Performance Monitor (APM) to support complex distributed structures like the Edge-Computing paradigm that scatters Cloud computation units in multiple layers between a central cloud and the users. However, the use of the semantic in the current monitoring landscape is poor, Cloud application, even when distributed into multiple physical locations, remains perceived as one flat network of components.

2.2.2 Collecting Traces in Jaeger Tracing

For traces, the storage backend application considered in this thesis is **Jaeger Tracing**. Indeed, as OpenTelemetry remains a young project there are yet few tools that are compatible with the pipeline. Still, Jaeger has been implemented multiple times in production by many companies and is also designated by OpenTelemetry as the reference implementation for storing traces. Figure 2.2 shows a part of a trace captured in an instrumented system: this trace is made of 63 latency measurements linked together, called **Spans** in the following.

These **Spans** are the building blocks of the model described throughout this thesis; they correspond to a unique and attributed latency measure for a given *Operation* on a given computing *Resource*. In figure 2.2, the spans displayed describe the interactions of six micro-services that serve a web page of a product in an online boutique application. A focus is made on one particular span of the *currency service* component. It has multiple attributes describing the action and the resource serving the request. These attributes follow the OpenTelemetry Semantic.

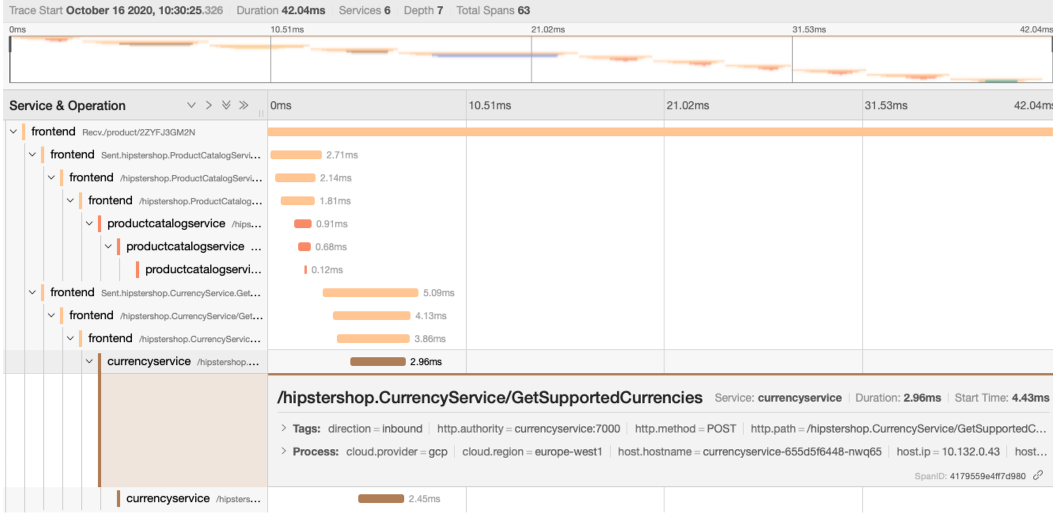

Figure 2.2: Part of a Jaeger Trace from an Online Boutique Application

Traces are built by Jaeger applications, the instrumentation clients only send latency measurements and the metadata containing a unique ID that enables the

aggregation of spans in a single trace. The main role of Jaeger is to aggregate spans into traces and to visualize them in its User Interface, it does not process the traces further. Therefore, by its nature, it is still heavily used in debugging and development scenarios but more rarely in a production setup. The process of investigating traces is a manual process requiring a human to perform the analysis.

Finally, Jaeger also exposes an API allowing users to request tracing data serialized in a format allowing further processing. Jaeger API uses the gRPC protocol with a protobuf encoding for a more efficient data format. Unlike REST APIs, protobuf APIs use a binary serialization format requiring an Interface Definition Language (IDL) describing the data objects exchanged between Jaeger and its client. Jaeger IDLs being public and Open Source, creating a client is still possible. Also, gRPC API allows streaming data objects from the Jaeger endpoint to the client.

As a result, by being a commonly deployed tool in industry, compatible with the OpenTelemetry pipeline, and having a powerful way to interface with custom software, the following of this thesis will integrate with Jaeger.

2.2.3 The Jaeger Analytic Library and its Limitations

Jaeger Analytics Library is another open source project[7] providing a set of functions for interfacing custom code with a Jaeger instance to manipulate traces as data objects. The core of this library consists of encoding each trace as a graph whose vertices are **spans**, it is used by Jaeger Tracing to discover dependencies among micro-services. Figure 2.3 provides an example of a simple trace encoded into a graph with the Jaeger Analytic Library. In this model, each vertex of the graph designates a span and all the metadata it holds. Edges are directed and designate a reference of a parent span to its child.

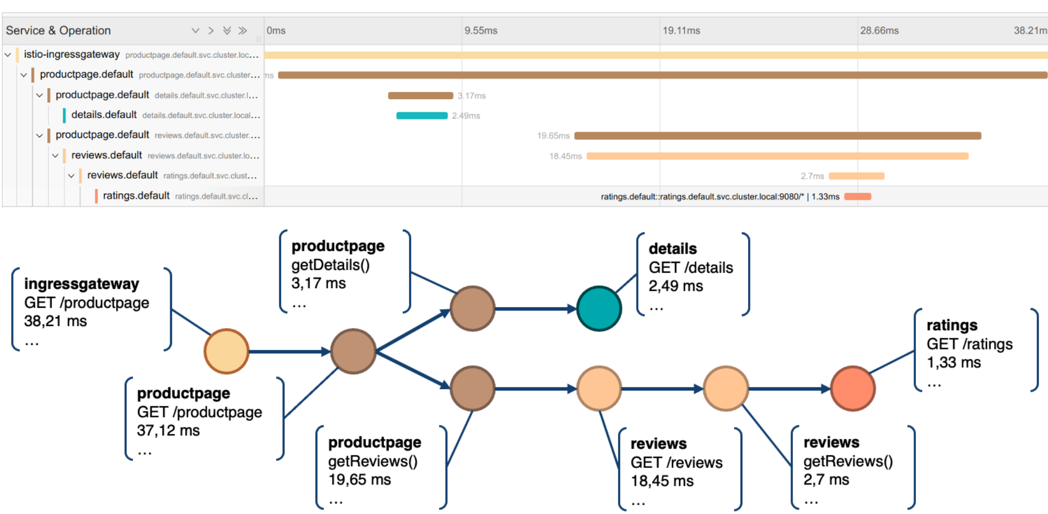

Figure 2.3: Graphical Representation of Jaeger Analytic Library Graph Encoding

[7]https://github.com/jaegertracing/jaeger-analytics-java A Java project providing a data model for traces in order to be manipulated for further data processing

With that model, each trace becomes a particular instance of a graph, and each graph is independent from each other. As a result, all graphs-related operations are not directly possible when analysing multiple traces. In addition, the model does not extract nor process spans' metadata.

In the following section, we present an approach that leverages OpenTelemetry semantic to build a property graph exhibiting interactions between components in a complex system. This will allow multiple traces to be merged in a single property graph, the resulting graph would show the overall application performance and components interactions.

2.3 Extracting a Structural Model from Traces

The Jaeger Analytic Library graph encoder creates one graph per trace and represents them as a Directed Acyclic Graph (DAG) of spans. In order to use traces to represent a wider view of the applications, multiple instances of traces are required and therefore a way to effectively compare spans must be found. However, spans are dense in attributes and comparing them is not an immediate process. Two spans may be characterizing the same operation done in two different instances of a microservice or, two different operations in the same microservice. For this reason, the OpenTelemetry semantic describes other entities that are related to spans: in particular, they describe *Resources*:

> A Resource is an immutable representation of the entity producing telemetry as attributes. For example, a process producing telemetry that is running in a container on Kubernetes has a Pod name, it is in a namespace and possibly is part of a Deployment which also has a name. All three of these attributes can be included in the Resource. (OpenTelemetry Specifications of the Resource SDK[8])

According to the definition, a resource may act as a common element shared by multiple spans and multiple traces. However, they do not appear in current implementations such as the Jaeger Analytic Library. Throughout this chapter, we propose an alternative to this library implementing a graph encoding model that allows to exhibit the multiple communications levels these *Resources* can have. Later chapters will demonstrate some usages of this model to solve Cloud specific performance issues like bottleneck identification or inefficient resource composition.

2.3.1 Identifying Common Elements to Aggregate Traces

In the definition of the *Resource SDK* of OpenTelemetry, a resource is defined as an immutable representation of the entity. It remains a single abstract entity that can take many shapes depending on the attributes it has. For instance, a resource with attributes `container.id` and/or `container.name` is effectively a container, a resource with attributes `host.name` and/or `host.id` is a virtual machine. In the

[8]`https://opentelemetry.io/docs/reference/specification/resource/sdk/` Page providing the specifications instrumentation libraries must adopt to follow OpenTelemetry Standard

same vein, attribute `k8s.pod.uid` uniquely identifies a Kubernetes *Pod*, attribute `k8s.node.uid` uniquely identifies a Kubernetes Node which is also a VM.

Nonetheless, attributes are not all mutually exclusives and, in a realistic cloud deployment, a resource may hold all of the following attributes `cloud.platform`, `container.name`, `k8s.pod.uid` and `k8s.node.uid`. A span in a trace having such attributes would be a latency measurement done in a particular container situated within a Pod, hosted in a particular Kubernetes node managed by a cloud provider. As a result, this resource materializes multiple underlying entities taking part in our cloud application, each being a potential cause of an issue or a failure. In our graph encoding method, we match all types of resources (e.g. *Pods*, *Nodes*, ...) from the attributes present in the resource. Therefore, for each span in a trace, we separate the performance measurement (the *Span*) from all the execution instances it is associated with (the *Resource*).

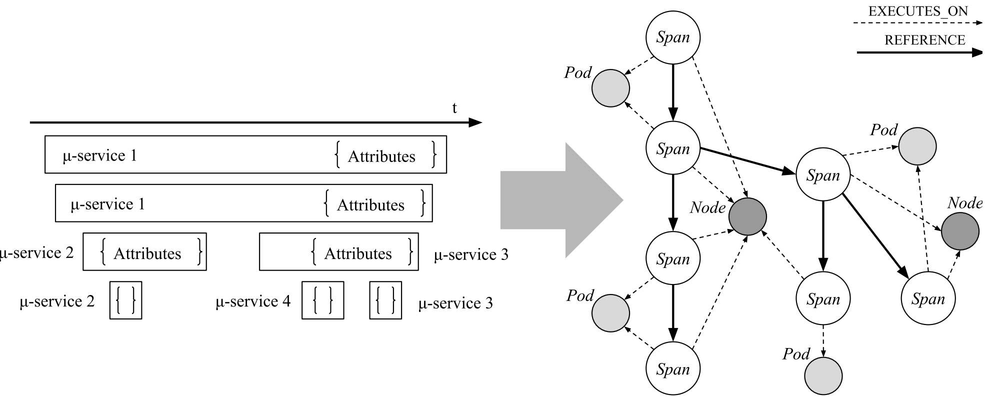

Figure 2.4: Transforming a Trace in a Property Graph.

Figure 2.4 demonstrates this graph encoding method on a small trace where: vertices can either be *Spans* (plain white circles in the figure) or instances of *Resources* (smaller circles with a grey-scaled filling). Edges can also have multiple types: in the figure, a plain arrow (→) represents a REFERENCE between two spans; a dashed arrow (⇢) represents a link between a latency measurement and its identified *Resources*, this edge is typed EXECUTES_ON.

According to the *Resource SDK*, resource attributes `k8s.pod.uid`, `host.id`, or `k8s.node.uid` are consistent across traces and allow to identify a resource taking part in multiple traces. These `ids` attributes are used to merge resources created from different traces into the same vertex in the property graph. Therefore, in a single graph we can start accumulating multiple traces and identify each component based on the flow of traces. Setting the resources in separated vertices allows to identify the common resources shared by multiple traces in our graph model. Still, these resources share no relationship with each other but some of them follow a containment relationship: for example, in a Kubernetes context, *containers* are always executed into *pods*, *pods* into *nodes*, *nodes* belong to *clusters*, etc. In the next section, we propose to enhance the model by including hierarchical relationship between these *resources*.

2.3.2 Modelling an Application Hierarchical Structure

These resources are inherently structured, the variety of services within a CSP or a Cloud orchestrator catalogue demonstrate how intertwined these services and resources are. In the scope of this thesis, we mainly focus on large-scale deployments of Kubernetes Clusters. While this work is tightly tied to Kubernetes platform, it remains a motivating example, and, in later chapters we apply the model to other cloud platforms. In the following list, we provide an overview of the various kinds of relationships that exists between resources in a large-scale Kubernetes deployment:

- *containers* require a *host* to be executed
- In the majors CSPs, *hosts* can belong to *Automatically Scaling Groups*
- Kubernetes *hosts* are called *nodes* in the Kubernetes DSL
- Kubernetes DSL defines *pods* as a collection of *containers* running within the same *host*
- Kubernetes DSL also defines logical resources entities that manage *pods*: *replica set*, *daemon set* and *stateful set* manage *pods* placements in the cluster. In the same flavour, *replica set* can be managed by *deployments*
- Most CSPs define the concept of *zone* (or *availability zones*) as a group of *data centre* tightly linked with each other within a *region*

Two main structures stand out from these observations for standard Kubernetes deployments:

- a logical hierarchy of the entities that follow the Kubernetes DSL $Pods \subset Replica\ Set \subset Deployments$ and $Pods \subset Stateful\ Set$
- a physical hierarchy of resources describing the location of microservices within a Kubernetes cluster: $Pods \subset Nodes \subset Clusters$

Whereas the logical hierarchy of Kubernetes entities does not have a particular value outside of this specific scope, the physical hierarchy that describes Kubernetes pod placement within a given hierarchy is a generic problem. Through this example we propose to use traces to model an automated process that can yield some unwanted scenarios.

The concept of hierarchy has been particularly studied in [Zafeiris 2018]. Authors propose an extensive review of hierarchical structures and their representativeness of real-world systems. Both of the previously identified hierarchies fit the definition of the **containment** (also named **nested**, **embedded** or **inclusive**) hierarchy. These hierarchies characterize structures in which entities are embedded into one another. Higher-level entities consist of and contain lower-level ones. Authors refer to the following definition for containment hierarchies: “larger and more complex systems consist of and are dependent upon simpler systems and essential system-component entities”. This definition also matches massive cloud deployments.

Now, Cloud Native Applications are scattered in multiple data centres, they bring computation resources to the edge. Voices assistants like *Amazon Alexa*, *Google Assitant* or *Apple Siri* eventually process the voice signal on the user devices. This allows serving the request with minimal latency while still depending on the cloud backend for actions. This suggests that more and more Cloud deployments will adopt this heavily hierarchical structure.

Most of major CSPs already adopted bigger scale Kubernetes deployment by proposing *Zonal Kubernetes Clusters* in their catalogue. These clusters leverage Availability Zones to make applications resilient to the loss of one data centre. The containment hierarchy of resources may be represented as: *Pods* $\subset$ *Nodes* $\subset$ *Zones* $\subset$ *Clusters*. Throughout the following we will consider these zonal clusters as motivating example to support our model; indeed while they are now adopted by cloud providers, they still have a range of open challenges. The main challenge we focus on is the ambiguity between the flat nature of the overlay network linking pods inside the cluster and the hierarchical structure of the network where all links do not have the same "cost".

From a mathematical point of view, the privileged representation of hierarchical structures are graphs with directed edges [Zafeiris 2018]. The hierarchy representation is straightforward, entities are represented by the vertices of the graph while dominance relationships between two entities are represented by an edge between these entities. In the initial graph encoding described in figure 2.4, the graph vertices already represent the resources. However none of the edges types define the resources hierarchical ordering. To add this containment structure to the model, we propose to add edges IS_CONTAINED to the embedded resources.

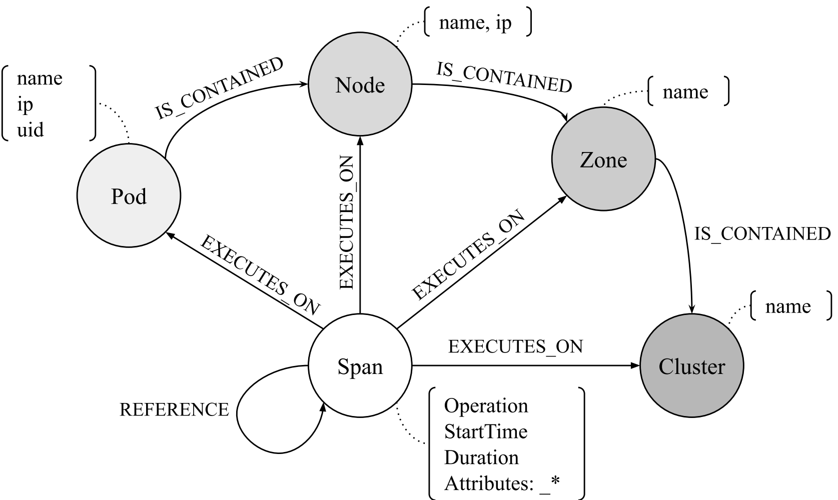

Figure 2.5: Transforming a Trace in a Property Graph.

Figure 2.5 describes the graph encoding **meta-model**: indeed, trivial graphs representing a 7-spans trace with two levels of hierarchy already lead to complex graphs. Instead, figure 2.5 represents the graph meta model, itself being a graph. The text valuies of each vertices in the meta model represent the labels the final graph can have, edges represent the relationship that can be found between two

vertices of different labels.

Figure 2.5 show the vertices labels and edges annotation that represent the trace-to-graph encoding mechanism in the context of a *Zonal Kubernetes Cluster*. The base graph model presneted for tracing is *Span* vertices linked together through *REFERENCE*s edges forming a DAG. With this graph encoding method, we go a step further by extracting the common identifiers of resources to create vertices that will bridge different traces in a bigger property graph. In the case of *Zonal Kubernetes Clusters* a four levels hierarchy can be extracted from its definition: $Pod \subset Node \subset Zone \subset Region$.

This resource extraction method makes steps aside from the OpenTelemetry semantic definition of a *resource*: instead of defining a resource as a unique entity, we define resource as an embedding of multiple sur resources. The proposed graph encoding of multiple resources following an order that forms a containment hierarchy. With this property graph model, we can use vertices labels and edges types to filter sub graphs that carry a special meaning. For instance, after accumulating multiple traces, extracting the sub-graph of all vertices linked by an IS_CONTAINED edge will provide a view of the way resources are structures forming the whole hierarchy of *Pods*, *Nodes* and *Zones* taking part in a cluster of an application. Also, extracting *Span* vertices adjacent to a particular resource will provide a filtering of all latency measurements of this resource.

2.4 Modelling Components Interactions

By leveraging OpenTelemetry semantic with the knowledge of Cloud architectures, we have been able to identify the physical components involved in the system from its traces. These components follow a containment hierarchy that describes the overall cloud-application structure. The graph meta-model defined in figure 2.5 highlights this physical organization of resources, but also another organization of these resources. Indeed, REFERENCEs relationships indicates how services compose together, and theses different compositions form a complementary organisation of resources also describing the cloud application. These resources follow an organization similar to information processing networks [Durugbo 2013]. In this section, we propose to use graph rewriting operations for extracting a hierarchical model out of a flow of traces.

2.4.1 Leveraging the Property Graph Model to Identify the Type of Communication

The REFERENCE edges can represent any level of function composition in a software system. For example, an OpenTelemetry instrumented function `f_parent` can call another function `f_child_1` within the same program to do local computations. Also, the same parent function can also use a Remote Procedure Call (RPC) to process a function named `f_child_2` executed on another machine. Both of these scenarios generate OpenTelemetry references from the parent to the child functions. However, one represents a local function call and the other represents a network call. Figure 2.6 illustrates this example, the two references are labelled to

identify which one is a local communication within the resource and which one is a network communications between resources.

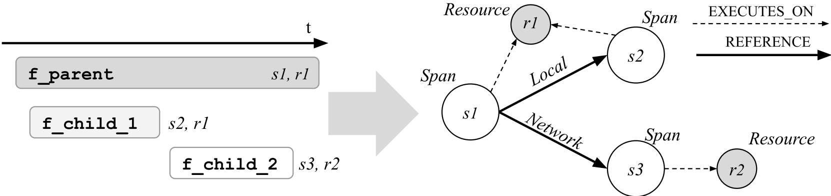

Figure 2.6: Example of Local Communications and Network Communication Between Resources

In our model, this difference does not appear directly on the REFERENCE edges. To identify if the communication is local or goes through the network traversing the graph is mandatory. In figure 2.7, Span vertices linked by a REFERENCE edge share a common resource. The REFERENCE edge demonstrates a communication that lives within that resource and do not cross its boundaries. Both spans *s1* and *s2* are linked to the same resource *r1*.

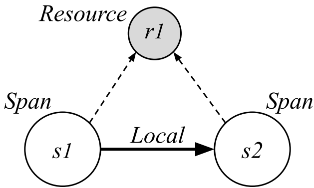

Figure 2.7: Graph Pattern Showing a Local Reference

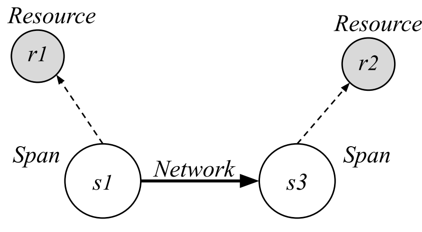

Figure 2.8: Graph Pattern Showing a Network Reference

On the contrary, in figure 2.8, span vertices linked by a REFERENCE edge have different resources. This represents a function call that crosses this resource boundaries. In practice, it corresponds to a RPC, a REST API call from a micro-service to another, it also fits the case of a database query. Therefore, not all REFERENCE edges characterize the same level of dependency within a distributed application.

These REFERENCE edges are only defined between *Span* vertices. Resources of the same level (e.g. *Pods*, *Nodes*, etc.) are not linked by any edge. However, paths exist for linking them. These paths linking two resources instances of the same level go back to the span vertices via an EXECUTED_ON edge, then find another span vertex via a REFERENCE edge that is executed on the target resource.

By identifying these paths between two resources of the same hierarchical layer, we can deduce which kind of cross-process communications are involved in the composition of service. Therefore, we can spot if the observed communication occurs within the same machine, crosses the network but stay within the cloud network, or if it goes through the internet. To identify the hierarchical level in which lives a REFERENCE edge, we use graph rewriting operations that project this REFERENCE edge at all the layers of the resource hierarchy.

2.4.2 Graph Rewriting Operations

With our custom graph encoding process, we are able to identify, by traversing the graph, at which layer of the containment hierarchy the communication is observed. The graph pattern from figure 2.8 expresses that the two resources involved, communicate over the network. Spotting all these patterns in a layer of the containment hierarchy would create a networking model of these resources. Graph rewriting operations Grantt the possibility of creating these edges based on the previously identified patterns.

In this section we first introduce formal definition of property graph and of graph rewriting. Then we present the rules that have been defined to sort communications in our model.

2.4.2.1 Formal Definition of Property Graphs and Graph Rewriting

Graph rewriting is a technique that defines algorithmic rules for creating a new graph out of an original one. These algorithmic rules can define both situations to apply the transformation and the transformation itself. Graph rewriting rules are usually noted $r : L \rightarrow R$ where L and R are two attributed graph, L designates the left-hand side of the operand and R the right-hand side. In this definition, both L and R are attributed graphs entities, they are noted $L = (V_L, E_L, \lambda_L)$ and $R = (V_R, E_R, \lambda_R)$.

In general, to define property graph, we use the definition of a multi-relational, attributed, directed graph noted $G = (V, E, \lambda)$. While there are multiple formal definitions of property graphs, we will use the definition of [Rodriguez 2015]. This definition is used for defining the Gremlin graph traversal language, used in the section 2.5.

- V is the set of the vertices of the graph G, each vertex has exactly one label l.
- $E \subseteq (V \times V)$ is the multi-set of directed edges of the graph G, edges also have exactly one label.
- λ is a partial function that represents properties in the graph. λ maps a pair made of an element of G (a vertex or an edge) and a string to any of the allowed attribute value of G. This partial function is noted $\lambda : ((V \cup E) \times \Sigma^*) \rightarrow (U \setminus (V \cup E))$. U refers to the universal set of values, λ partial function allow linking to any values minus the element of G (noted $(V \cup E)$).

This $r : L \rightarrow R$ rewriting rule is applied on the original graph by searching occurrences of the pattern L, which is deleted and then replaced by the pattern R. Deleting some of the elements in a graph may lead to cases where a vertex is deleted but edges pointing to this vertex remains untouched. After such a transformation, these edges would not point to any other vertices in the graph, they are called *suspended edges*. Two mains approaches have been defined to handle the cases of suspended edges: **Single Pushout (SPO)** and **Double Pushout (DPO)** [Parisi-Presicce 1993].

- Single Pushout (SPO): it allows to add, delete, merge or clone vertices or edges in an attributed graph, deleting any suspended edges.
- Double Pushout (DPO): it allows to add, delete, merge or clone vertices or edges in an attributed graph, but blocks if any suspended edges is encountered.

The graph L is designated as a pattern graph and can also be expressed as a pattern matching sequence. In [Francis 2018], authors present the Cypher language implemented in the Neo4J graph database. Unlike the Gremlin language that describes graph traversal, Cypher focuses heavily on pattern identification in graphs. In this work, authors define graph pattern Π as a multiple path patterns noted π,. They define a path pattern noted π as a sequence of vertices pattern, noted χ, and edge patterns, noted ρ.

$$\Pi = \{\pi_1,\ \pi_2, \dots \pi_m\}$$
$$\pi = \chi_1\ \rho_1\ \chi_2\ \rho_2 \dots \rho_{n-1}\ \chi_n$$

Using pattern matching sequences would provide a more general formulation of graph L that can be applied to all kind of resources existing in a cloud environment. Consequently we provide a definition of the patterns χ and ρ that fits our initial definition of property graph. Indeed, in [Francis 2018] vertices can have multiple labels whereas in the definition of property graph taken from [Rodriguez 2015] vertices can only hold one label. Hence, a node pattern χ is defined as a triple $(a, \mathcal{L}, \mathcal{P})$ where:

- a is the name of the vertex in the pattern, the name may possibly be empty. When it is not it allows to reuse this node in later stages of the matching sequence.
- $\mathcal{L}$ corresponds to a constraint on the label of the vertex on which the pattern χ is applied. It may be empty to match any vertex with any label.
- $\mathcal{P}$ corresponds to constraints on the properties of the vertex on which the pattern χ is applied.

Edges patterns ρ are defined as the tuple $(d, a, \mathcal{L}, \mathcal{P}, \mathcal{I})$ where:

- $d \in \{\rightarrow, \leftarrow, \leftrightarrow\}$ specifies the direction of the edge.
- a is the name of the edge. Like for vertices, names allow to reuse the edge whose pattern has been matched against in later stage of the matching sequence.
- $\mathcal{L}$ corresponds to a constraint on the label of the edge on which the pattern ρ is applied, it may be empty to match any vertex with any label.
- $\mathcal{P}$ also corresponds to constraints on the properties of the edge on which the pattern ρ is applied.
- $\mathcal{I}$ is either nil or (m, n). When defined, $m, n \in \mathbb{N}$ corresponds respectively to the minimum and maximum number of edges to traverse.

2.4.2.2 Graph Rewriting Rules for Finding Network Communications

Graph rewriting operation can be used to find the patterns that have been identified to represent network communications between resources. The result of the operation would be a graph representing only the communications between resource of the level of hierarchy considered. Figure 2.9 provides a graphical representation of the graph rewriting rule applied to *Pods*. The first line shows on the left side of the operand the graph L being the pattern that will be searched into the graph. On the right side the graph R is represented. It is the result of the process (the creation of the new edge) applied on this pattern. The second line provides an example of this transformation on a graph. The left part of this the second line reprents the trace encoded as graph (like the one presented in figure 2.4 but only with *Pods* being displayed as resources for the sake of clarity). It is the starting point of the stransformation. On the other side the result of the graph rewriting operation is presented. The goal of this rewriting process is to lessen the number of vertices in the graph to better focus on resource interactions, materialized by a PROJECTED_REF edge.

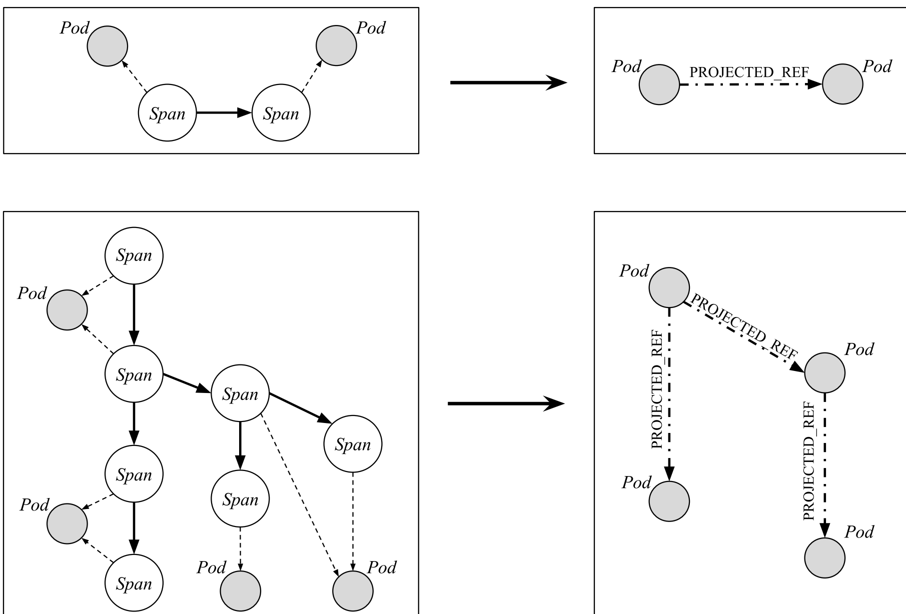

Figure 2.9: Graph Rewriting Approach to deduce resources dependencies (applied to Pods) based on a Simple Pushout operation.

For the purpose of this transformation, the SPO is convenient because it deletes the Span vertices and the EXECUTES_ON and REFERENCE edges once the PROJECTED_REF has been created. Therefore, after the rewriting process, no nodes labelled *Span* remains nor edges labelled EXECUTES_ON. The final graph is only made of resources vertices with IS_CONTAINED and PROJECTED_REF edges.

The rule illustrated in figure 2.9 is specific for *Pods*, but the rewriting rules remains the same for any other pair of resource vertices belonging to the same

abstraction layer of the resource containment hierarchy. Figure 2.10 details how the rewriting rule simplifies the graph by providing intermediates steps in the pattern matching process. In a first sub-step, the figure highlights the patterns of network communications and discard the local communications. The pattern identified in the first step of the graph rewriting notation from figure 2.9 appear in colored groups in figure 2.10. Each colored group represents an instance of the network communication pattern. the second sub-step shows the discarded *Span* vertices and demonstrates the creation of the *PROJECTED_REF* edge between the *Pod* vertices.

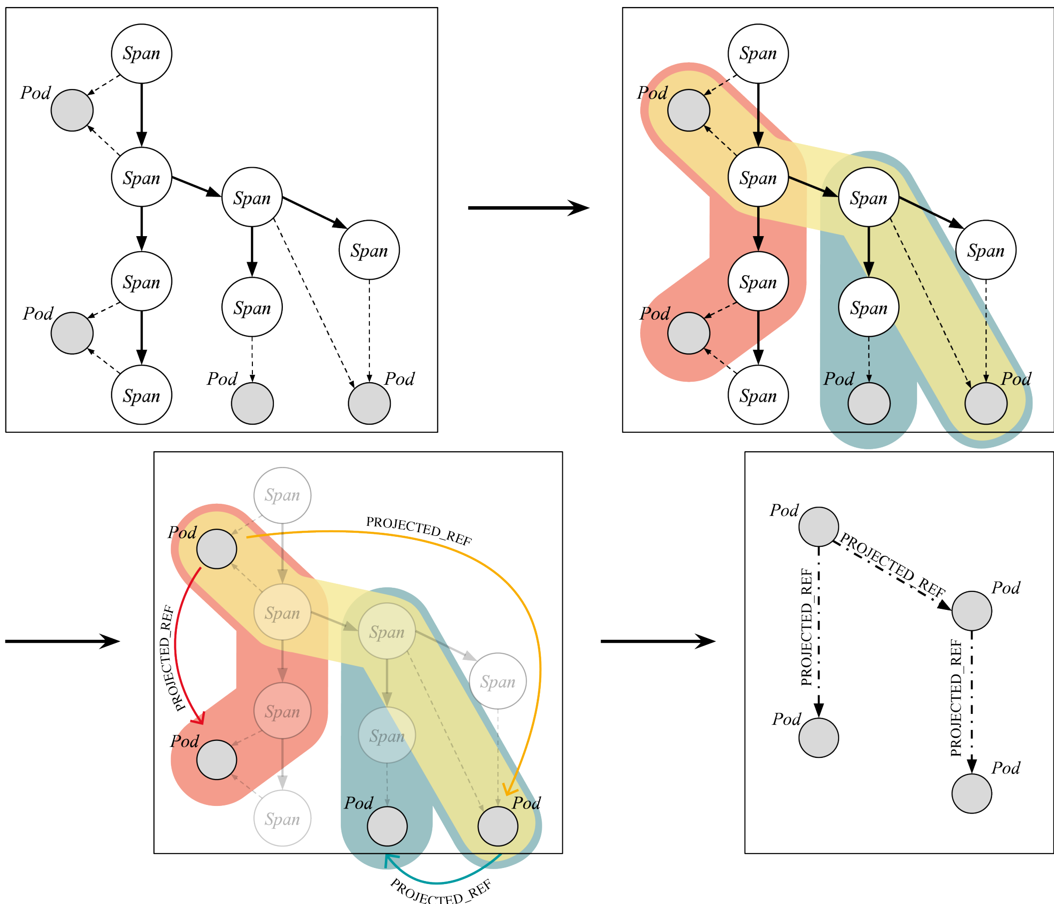

Figure 2.10: Graph Rewriting process explained step by step

This rewriting rule can be expressed with a graph pattern matching representation as defined in [Francis 2018]. Equation 2.1 defines the pattern π as being a parameterized list of vertices and edges of the graph L and returns the nodes r_1 and r_2 matching this representation. Pattern rule π is expressed as a function taking a *ResourceLabel* as parameter to specify the pattern where $ResourceLabel \in \{Pods, Nodes, Zones, \ldots\}$. A visual representation is provided in figure 2.11.

$$\begin{aligned}\pi(\mathit{ResourceLabel}) = &(r_1, \mathit{ResourceLabel}, \varnothing),\\ &(\leftarrow, \text{nil}, \text{EXECUTES_ON}, \varnothing, (1,1)),\\ &(\text{nil}, \text{Span}, \varnothing),\\ &(\rightarrow, \text{nil}, \text{REFERENCE}, \varnothing, (1,1)),\\ &(\text{nil}, \text{Span}, \varnothing),\\ &(\rightarrow, \text{nil}, \text{EXECUTES_ON}, \varnothing, (1,1)),\\ &(r_2, \mathit{ResourceLabel}, \varnothing)\end{aligned} \tag{2.1}$$

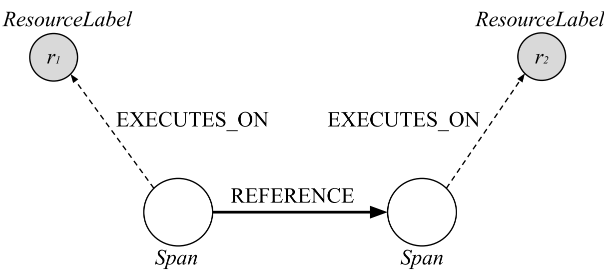

Figure 2.11: Visual Pattern representing equation 2.1

With the pattern matching rule defined in equation 2.1, we obtain the resources vertices of label *ResourceLabel* designated as r_1 and r_2 that can later be used to create an edge labeled *PROJECTED_REF*. Therefore, when these *PROJECTED_REF* are added to the graph for all the pattern of the initial graph, vertices labeled with *ResourceLabel* are linked together by edges modelling their network references only.

2.4.3 Building a Hierarchical Property Graph

After the rewriting process, we obtain a new graph for each level of the containment hierarchy. These new graphs express the communications between their entities for a given set of traces. Merging these graphs back in the original graph would add the edges *PROJECTED_REF*. Figure 2.12, represents the property graph with all the new edges and vertices. In this figure, resources vertices have been grouped by labels: a group of pod vertices has been named *Pods* and a group of node vertices has been named *Nodes*. Also some *EXECUTES_ON* edges have been omitted between the span vertices and the node vertices for the sake of readability.

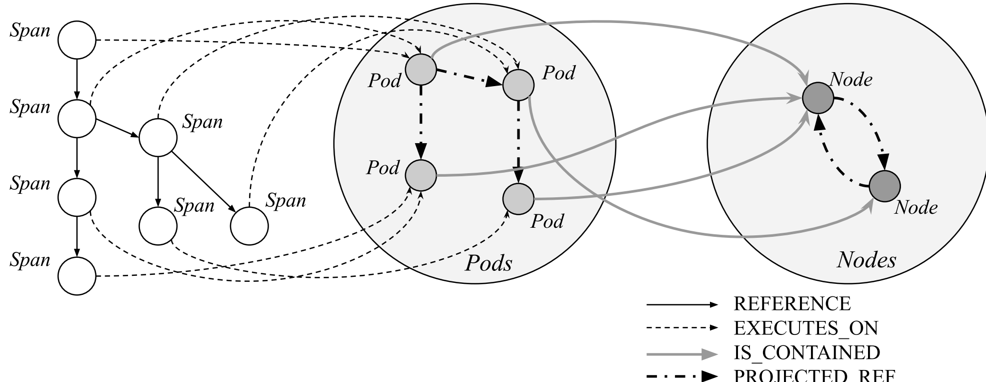

Figure 2.12: Hierarchical Graph Representation.

Whereas the property graph from figure 2.12 is not a hierarchical structure by itself, it exhibits a multi-layers structure. Indeed, in [Drewes 2000], authors propose to extend the SPO and DPO rewriting rules to hierarchical graphs. In this work, authors define a hierarchical graph as a graph of graphs: a DAG whose nodes are graphs and edges are morphisms between elements of these graphs. These morphisms between elements of the graphs are called *frames* in [Drewes 2000].

The graph representation in figure 2.12 matches this definition. Nodes of the hierarchical graphs are each of the graph obtained by the rewriting operation. In this designation, *Pods* is a graph where its vertices are labelled *Pod* and its edges are typed PROJECTED_REFs. The same applies to *Nodes*, and all the layers of resources that can be extracted from traces. The *frames*, showing the hierarchical relationship between entities of each layers, are materialized by the IS_CONTAINED edges: $Pods \subset Nodes \subset Zones \subset Clusters$.

As a result, traces, which are flat graphs where abstraction levels are hidden, have been turned, by this graph encoding method, in a multi-level location-aware model that highlights the composition of service and resources. This encoding model will be used in later chapters to process traces and detect issues specific to distributed systems communicating over a multi-layer network. The next section, details the implementation of this model.

2.5 Implementation

Both graph encoding logic and rewriting operation have been implemented in the *Scala* programming language. This implementation is based on the Jaeger Analytics Library presented in section 2.2.3. Like the original, this work decodes data from Jaeger gRPC API but transforms this data to implement the model and the rewriting operation. The choice of the *Scala* programming language was natural as *Scala* is compatible with the Java ecosystem and also has a wide range of libraries related to graph processing. In addition, *Scala* uses a functional programming approach and is compatible with *Spark*, a technology enabling massive and parallel data processing.

In this section we present the implementation of our online processing pipeline

for parallel trace computations; this implementation has been used to process data and structure our hierarchical property graph. To develop our pipeline we used the Scala data-processing platform named *Polynote*[9]. This platform is a Notebook engine capable of executing code written in Scala and in Python, it provides an environment allowing to quickly build our pipeline on a real data flow. In this section we present the four stages of the pipeline created in Polynote: reading data from a Jaeger gRPC endpoint, applying the model to create graphs, and finally, applying the rewriting process on the graph to generate a hierarchical structure.

Figure 2.13 represents the pipeline: the vertical left part is standardized by OpenTelemetry and corresponds to a normal Jaeger Deployment. The horizontal part of the pipeline represents what was added to support the graph operations required to maintain the model. The two steps pipeline in Polynote corresponds to the Scala code, all these steps can be parallelized to process Jaeger Data more efficiently. The final model is hosted in a Graph database in which multiple scala worker can write simultaneously.

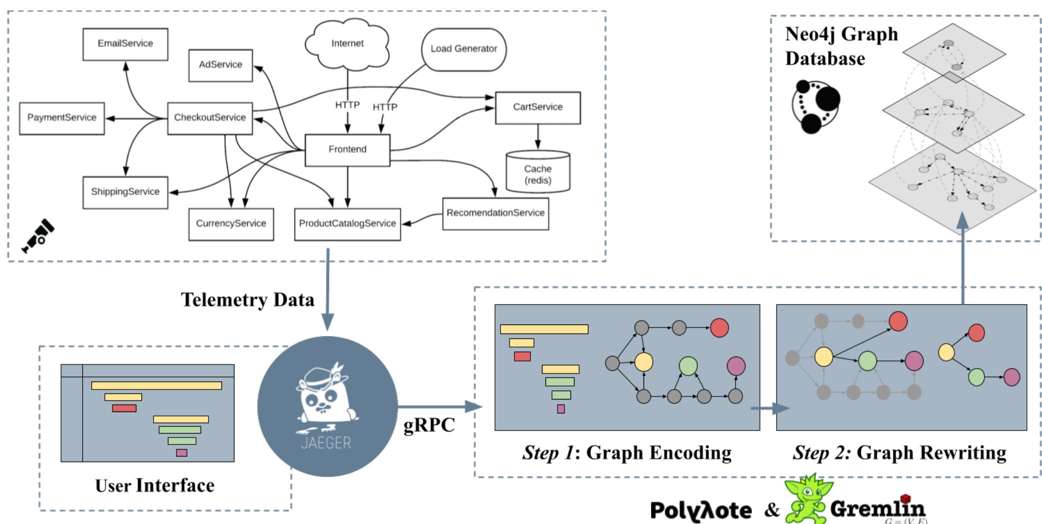

Figure 2.13: Complete Telemetry Processing Pipeline

The *Polynote* notebook Scala code are presented in appendix B, in the following we refer to some sections of the code presented with a focus on key elements. Still, the Scala language can be verbose when dealing with nested data structures and with Java libraries, therefore the code snippets presented may refer to code some section that are only detailed in the appendix.

2.5.1 Extracting Data from a Jaeger gRPC Endpoint

The Jaeger Tracing tool exposes its data through a gRPC API; unlike REST APIs, gRPC APIs are made of a binary data flow, using the *Protocol Buffers (protobuf)* serialization format. It makes the decoding operation more complex to implement.

[9] `https://polynote.org` A Scala Notebook engine open sourced by Netflix

For our pipeline we used the compatibility with the Java ecosystem to use the decoding functions already defined in the Jaeger Analytics Library.

Once connected to the gRPC endpoint, Jaeger only provides a flow of *Span* data objects (designated by the class renamed `ProtoSpan` in all the code extracts). While these objects embed the *Resource* attached to the span, *Trace* objects are still to be built. The first step of our pipeline is, therefore, to rebuild traces based on the field `traceid` present in the Spans. Indeed, Jaeger provides through its endpoint, all spans it collects without ensuring that they belong to a known trace. A naive approach would be to collect all spans and group them based on the field `traceid`. However, there are edge cases, especially when querying batches of *Spans*, having a missing span from a trace would eventually break a DAG of spans into multiple separated components. So, before deriving our model from Jaeger data we first need to ensure that traces are complete.

In order to eliminate partial traces from span batches, we applied two checks before providing the trace to our custom graph encoding method. The first one checks if there is no missing Span objects; it checks that all `spanId` mentioned in the references field exist within the group of spans forming the trace. Then we check if the DAG of spans only has a single root element. We consider a root span a *Span Vertex* with no incoming REFERENCE edges.

In listing 1, we provide a fragment of the logic that decodes gRPC flow and build a data model that will later be used for encoding data as a property graph. At this stage, we ensure data consistency before encoding a property graph. The function `Trace.of(pSs: List[ProtoSpan])` aims to process a list of protobuf encoded Span objects that have already been grouped by `traceId`. This function return type is `Either[List[ProtoSpan], Trace]`, it means that the function will either be able to process the list of Spans and return a trace, or it will fail and return back the data without further processing. In our Jaeger client implementation, unprocessed data is cached for being added to the next batch, as a result partial traces, distributed into multiple batches of spans will still be processed by our client.

In this stage of the pipeline, we still follow the Jaeger data model; therefore *Resources* are still a single entity with a variety of semantic attributes. Also, each span has its own resource attached to it. In the next section, we will focus on extracting more data from these resource field and apply our meta model and build an instance of a property graph per traces.

2.5.2 Property Graph Encoding

With our Jaeger client, we query a batch of thousands of spans and aggregate these spans in traces while eliminating most edge cases. In this step, we process these traces to create an instance of a graph per trace. These graphs are merged in a later step of the implementation.

This graph encoding process has been implemented with the Gremlin graph query language [Rodriguez 2015] on an abstract implementation of property graphs. Gremlin can have multiple underlying implementations: it can either be an in-memory graph or graph databases. Both of these cases have been implemented in the context of the graph encoding:

```scala
// Renaming the class Span provided by the Jaeger Analytics Library
// as ProtoSpan to not interfere with our definition of Span
import io.jaegertracing.api_v2.Model.{Span => ProtoSpan}

// Definition of Trace as an aggregation of Spans
case class Trace(spans: List[Span])

// Companion Object providing static methods for the class `Trace`
object Trace {
  def apply(spans: List[Span]): Trace = new Trace(traceId, spans)

  // For a given collection of protobuf spans, either provides a
  ↪  trace object or return back the given collection when the
  ↪  trace is not consistent
  def of(pSs: List[ProtoSpan]): Either[List[ProtoSpan], Trace] = {

    // Convert each ProtoSpan in a Span
    val spans: List[Span] = for { ps <- pSs } yield Span.of(ps)

    val spanReferences = for {
      span <- spans
      ref  <- span.references
    } yield ref                // Extract all References

    // Check if all references are known within the trace
    val isCompleteTrace = spanReferences forall {
      spanRef => spans exists {
        span => span.spanId == spanRef.spanId
      }
    }

    // Find the spans without any ancestors within the reference
    val rootSpans = spans filterNot {
      span => spanReferences.map(_.spanId).contains(span.spanId)
    }

    // Return only the `Right` case when the trace is consistant
    if (isCompleteTrace && rootSpans.length == 1) {
      Right(this.apply(spans))  // Normal case: instanciate a Trace
    } else {
      Left(pSs)             // Error case: leave parameter untouched
    }
  }
}
```

Listing 1: Trace Companion Object Ensuring Trace Consistency

- **In-Memory Graph:** They make computations faster for small-scale graphs, and, each trace being made of up to a hundred spans in the worst case, a graph instance would still fit in memory. **Tinkerpop Graphs** are the standard implementation of property graphs using Gremlin, they are lightweight and fast to traverse compared to graph databases. Still, there are not any visualization tools associated to this implementation.

- **Graph Databases:** They are capable of storing and querying thousands of vertices and edges, and are particularly efficient at high scale. Their performance, however, on a smaller graph, are not good enough to provide an almost real-time processing. We used **Neo4j** as a graph database. Neo4j addresses the drawback of in-memory graph: it comes with an efficient graph browser and a query language based on the pattern matching.

Therefore, all graph processing presented in this section can either be run on in-memory graph or on graph databases. Creating a graph encoding process with an abstract backend has granted us granularity for our experimentation and research. In a first stage, we used the graph database backend to visualize the graph, and to quickly iterate over our graph implementation. Then we use in-memory graph when visualizations are not required any more and when the constraints were performance.

The first stage of the graph encoding is to identify all the resources involved in a trace. As we focus on Zonal Kubernetes Clusters, the following examples will only cover the resources present in these clusters. To model the different types of Resource vertices with their different attributes, we used Scala Algebraic Data Types. Algebraic Data Types can be compared to enumerations in other languages. Listing 2 present an abstract type (`trait` in Scala) called `ResourceKind` which can be instantiated in one of the following objects: `Pod`, `Node` or `Zone`. Later we use OpenTelemetry attributes to instantiate all these resources when the corresponding labels are encountered.

```
sealed trait ResourceKind

case class Pod(name: String, ip: String, uid: String) extends
↪  ResourceKind
case class Node(name: String, ip: String) extends ResourceKind
case class Zone(name: String) extends ResourceKind
```

Listing 2: Algebraic Data Types Modelling Resource Matching

Figure 2.14 shows the result of the graph encoding method detailed previously. In this visualization we consider a Neo4j backend where multiple traces have been accumulated in a single graph instance. Still, not all vertices and edges have been displayed for the sake of clarity, mauve-coloured vertices represent spans, orange vertices represent pods. The green vertices represent *Operations*, *Operation* vertices were initially used in early definitions of the model but finally were discarded

to focus on the physical location of resources. The graph visualization still provides an expressive view that illustrates that each spans share common attributes materialized with specific vertices.

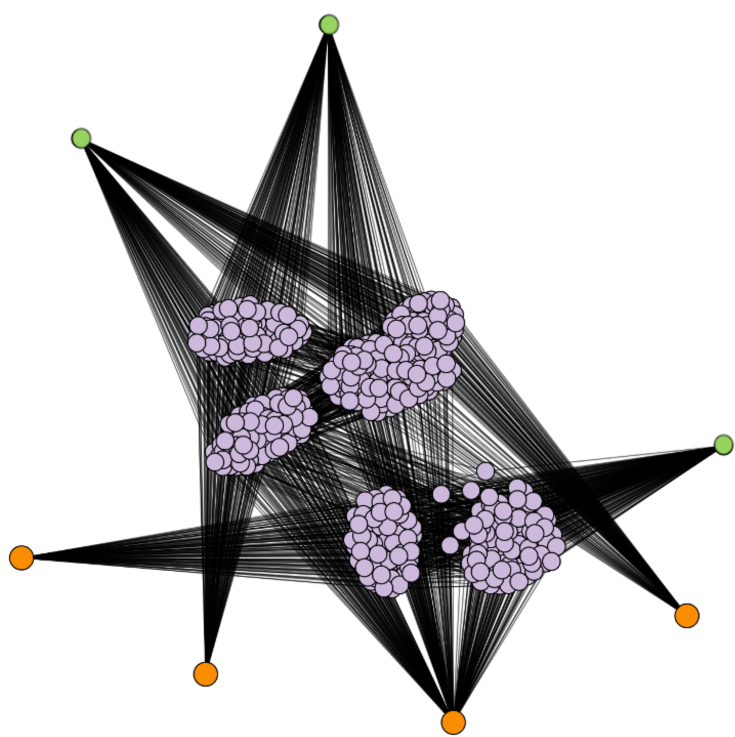

Figure 2.14: Aggregation of Multiple Traces Within a Single Graph Instance

2.5.3 Graph Rewriting Operations

In that section, we focus on the implementation of the graph rewriting operation. Graph rewriting operation has been implemented by integrating Gremlin functions within the Scala code base. It takes the form of a chain of Gremlin functions that are called on a particular graph tranversal object `g` already defined in the code. The Gremlin queries rewrites the graph in-place, therefore, there is no creation of a new graph at the end of the rewriting process. So, by directly calling Gremlin functions on the graph object, we have been able to minimize the computation resources required to perform this operation. The function returns the list of the edges added to the graph.

Listing 3 provides the code of the rewriting function: some sections of the code have been highlighted. The first one, from line 9 to line 13, shows a Gremlin query are directly integrated into the logic of the code. This query represents the Gremlin language direclty being integrated into the program logic: the result of the query is directly mapped as a Scala value. This query returns a list of pairs of Spans linked by a REFERENCE edge. These spans are later processed to find the resources they are executed on (lines 22 and 23 in the same function). The second

```
1  def projectDependencyOn(labels: String*)
2                         (implicit graph: ScalaGraph): List[Edge] = {
3    import TraceMetaModel._
4    val g = graph.traversal
5
6    // Gremlin Query to find all REFERENCES edges and get the Source
7    // and Dest vertices
8    val clientServerSpanVertices =
9      g.V().hasLabel(SpanLabel).as("srcSpan")
10       .out(References)
11       .hasLabel(SpanLabel).as("dstSpan")
12       .select("srcSpan", "dstSpan")
13       .toList
14       .map(_.asScala)
15
16   val dependentResourcesEdges = for {
17     label <- labels
18     v     <- clientServerSpanVertices
19   } yield {
20     val srcSpanV: Vertex = v("srcSpan").asInstanceOf[Vertex]
21     val dstSpanV: Vertex = v("dstSpan").asInstanceOf[Vertex]
22     val srcResOpt: Option[Vertex] =
       ↪  g.V(srcSpanV).out(ExecutesOn).hasLabel(label).headOption
23     val dstResOpt: Option[Vertex] =
       ↪  g.V(dstSpanV).out(ExecutesOn).hasLabel(label).headOption
24
25     // Uses Scala Pattern matching to know whether the communication
       ↪  local or going through the network.
26     (srcResOpt, dstResOpt) match {
27       case (Some(srcRes), Some(dstRes)) if srcRes != dstRes =>
28         Some(srcRes --- "PROJECTED_REF" --> dstRes)
29       case _ => None
30     }
31   }
32
33   dependentResourcesEdges collect { case Some(v) => v } toList
34 }
```

Listing 3: Function rewriting the graph in place

highlighted part of the code snippet, lines 27 and 28, represents the conditional creation of the edge `"PROJECTED_REF"`, which relies on scala pattern matching. With this implementations, the rewriting part requires only to traverse the graph once to list the source and destination spans of each REFERENCE edges. All other gremlin queries in the function directly address indexed entities of the graph: `g.V(srcSpanV)` reference a vertex by its ID and do not require to traverse the graph. On the contrary `g.V()` queries traverse the graph until their stopping condition is met.

Figure 2.15 shows the previous graph before and after the rewriting operation. As the function is applied in-place, the right graph is a sub-graph of pods linked by the PROJECTED_REF edges.

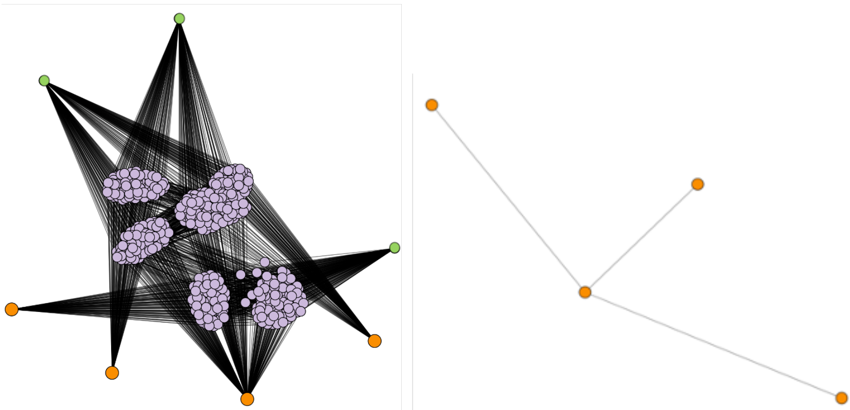

Figure 2.15: Rewriting Operation on the Graph at the *Pods* level

2.6 Conclusions

In this chapter we provide an overview the tracing ecosystem with the recent initiative OpenTelemetry. OpenTelemetry's role is to standardize distributed tracing by proving a semantic for traces as well as a default implementation suitable for Cloud deployments. In the past years, this project matured and gained an increase visibility in the field of Cloud technologies, several established cloud monitoring projects adopted OpenTelemetry format for traces like *Linkerd*[10] or *Cilium*[11]. And while this telemetry data is increasingly present in Cloud ecosystem trace useage is still poor in State-of-the-Art Cloud monitoring solution.

In this thesis, we identified the opportunity of this emerging data-source with a well-defined semantic to create a runtime model of a Cloud Native Application. Based on this data source we propose a model that uses traces to transform a flat networking model of components in multilayer hierarchical graph that focuses on resource location. The proposed model is generic as it does not rely on a particular

[10] `https://github.com/linkerd/linkerd2/pull/6188` Replacmeent of OpenCensus Tracer by OpenTelemetry

[11] `https://isovalent.com/blog/post/2021-12-release-111` Addition of OpenTelemetry Tracing into Cilium Service Mesh

Cloud implementation, it can model each of the entities part of the OpenTelemetry Resource semantic definition. This model has been inspired by Edge-Computing and Fog-Computing layered networking model and represents components interactions in a non-homogenuous network. Even in traces, resource location is partially hidden as state-of-the-art tools put a heavier focus on representing the application as a whole, not displaying the segregation among data centres, and other hierarchical layers. With the proposed model, we leverage OpenTelemetry semantic for identifying resources and tracing data to maintain a model at runtime based on a continuous data-flow of traces.

Then, an implementation of this model has been detailed capable of ingesting a flow of traces at runtime and maintaining a property graph of components interactions. This implementation is based on the Scala programming language, on Tinkerpop In-Memory graphs for quick Graph rewritting operation and finally on Neo4j for maintaining a graph model aggregating multiple traces. In the next chapters we apply this model on two different Cloud architectures: Chapter 3 describes Zonal Kubernetes Clusters Applications and Chapter 4 focuses on AWS global applications.

Chapter 3

Identifying Inefficient Service Composition with Flow Hierarchy Index

Contents

3.1 Introduction **59**

3.2 Modelling a System With Hierarchies **61**

3.2.1 Definition and Subtypes of Hierarchy 61

3.2.2 Measuring Imperfect Flow Hierarchies 62

3.2.3 Cycle Identification 64

3.3 Detecting Inefficient Service Composition **66**

3.3.1 Application to the Hierarchical Property Graph 66

3.3.2 Proof of Work on a Sample Cloud Application 68

3.4 Implementation **71**

3.4.1 Designing a Multi Layers Platform with Zonal Kubernetes Cluster 71

3.4.2 Getting OpenTelemetry Traces With Network Level 72

3.4.3 Computing the Flow Hierarchy Metric 75

3.4.4 Results 77

3.5 Conclusions **79**

3.1 Introduction

In the previous chapter we demonstrated that, by encoding traces as a property graph, we can decline them into a graph of communications sorted by abstraction layers. A trace, originally represented as a DAG of spans, can also represent interactions between VMs, containers, availability zones or any physical or logical entity identified in the trace. These graphs are obtained by leveraging the specificity of traces: the relationships between measurements. These relationships, observed between spans, are "projected" to the resources entities associated with them. Once projected, we observed that, for some traces, the initially acyclic trace graphs, can fall back on themselves after the "projection".

With this model, it became possible to address a challenge encountered by the teams developing and maintaining the *Djingo* Application. Indeed, the microservices of the application have been scattered in multiple data centres across Europe. Some performance anomalies have been observed in the application and the monitoring team was searching for a mean to express these undesirable communication patterns based on the global latency of the request.

In concrete terms, when a cloud application serves a request, the network communications cycle get through hosts and translates in an inefficient composition or placement of the microservices. To comply with the Non-Disclosure Agreement related to the *Djingo* application, the following chapter will consider the case of Zonal Kubernetes Clusters which exhibits a similar hierarchical pattern among the resources executing the microservices. In the case of a Zonal Kubernetes Cluster, observing network communications cycling between availability zones would result in a more expensive Cloud bill. Considering a global application scattered into multiple data centres, observing the communications of the services cycling through data centres would likely have a greater impact on both the global latency and on the cloud bill. In this, chapter we propose a method for processing the hierarchical property graph defined in chapter 2 that highlights the previous use cases through a generic approach.

This approach is based on the concept of flow hierarchy studied by [Luo 2011]. A particular focus is put on the flow hierarchy metric (noted h in the following). This metric is computed on sub-graphs extracted from our hierarchical property graph. The concept of flow hierarchy is associated with directed graphs and finds its usage mainly for describing self-organizing networks. The flow hierarchy metric h aims to detect and measure the extent to which all the edges in the graph follow a holistic overall "underlying direction" [Luo 2011]. Throughout this chapter we use the projections of OpenTelemetry traces onto the different resources to represent our self-organized networks of cloud resources.

This chapter is structured as follows:

Section 3.2 presents the theoretical concepts behind the concept of flow hierarchy with a focus on imperfect flow hierarchies and their capabilities of modelling a distributed application. Then, we present the flow hierarchy metrics which quantify the extent to which a directed graph follows an underlying direction. Finally, we detail the approach for computing this metric.

Section 3.3 presents the use of the flow hierarchy metric in our model, the adaptations of the metric to the hierarchical model and finally provides a theoretical study on a sample Cloud Application.

Section 3.4 presents the Proof-of-Concept Cloud deployment executing the previous Cloud Application on a physically distributed platform. We detail both the organization of the instrumentation and the integration of the computation of the flow hierarchy metric in the pipeline.

3.2 Modelling a System With Hierarchies

3.2.1 Definition and Subtypes of Hierarchy

In [Zafeiris 2018], authors provide a study on the hierarchical structures that model common complex systems in nature. For example, social structures of animal groups, human virtues, or even the structure of a computer program can be represented as hierarchies. Authors define a hierarchy as a structure whose entities belong to a system that can be classified into levels. These levels follow an order and elements of a higher level have an influence over entities of lower levels. Through this definition, authors set the core of the hierarchy definition being entities controlling, or having an influence, on the behaviour of others.

> A system is hierarchical if it has elements (or subsystems) that are in dominant-subordinate relation with each other. A unit is dominant over another unit to the extent of its ability to influence behaviour of the other. In this relation, the latter unit is called subordinate. ([Zafeiris 2018])

While this definition does not characterize how hierarchical a system is, it expresses the hierarchy through its elements relationships. In a more abstracted approach, a hierarchy can be seen as a generic structure in which levels are asymmetrically ranked. The way ranking is established can vary. Authors provide subtypes of hierarchies to classify how the ranking occurs:

The ordered hierarchies correspond to an ordered set. This hierarchy does not consider the network linking elements of the system. Each element of the system is associated to a value that defines its rank in the set.

The containment hierarchies correspond to entities embedded into each other like the Russian Matryoshka Dolls, the tree of life or resource placement in a Cloud as defined in chapter 2. This hierarchy materializes by a "belonging" relationship like *Foxes* ⊂ *Canidae* ⊂ *Carnivora* or *Pods* ⊂ *Nodes* ⊂ *Zones*. This type of hierarchy can be represented by a pure tree dendrogram.

The flow hierarchies are materialized by a directed graph whose vertices are entities of the system and the directed relationship corresponds to the flow of orders. They are also called *control hierarchies*, and can represent flow of order between armies or politicians. A *perfect* flow hierarchy is often model as a DAG, however, many examples of imperfect flow hierarchies can be encountered to model real-world problems.

Most major studies on hierarchical structures have focused on containment hierarchies [Anderson 1972, Clauset 2008, Sales-Pardo 2007]. In another hand, flow hierarchies have remained rather uncovered [Luo 2011]. Still, these two subtypes of hierarchies are not mutually exclusives: a directed graph may be modelled after both a containment and a flow hierarchy. This is the case for most software systems: a software is designed by writing functions, usually grouped in packages or libraries

which call each other in a directional flow. This example exhibits both the containment and the flow hierarchies: the first one describing a static architecture of the software through a tree of subsystems. The second one, the flow hierarchy, corresponds to the directional flow of these subsystems calling each other to assemble the software output.

In practice, keeping a software code organized has always been a challenge, in particular for large code bases: this problem has traditionally been encountered and studied in the java ecosystem [Breivold 2008, Al-Mutawa 2014]. These studies focus on circular and inter-modules dependencies on Java code bases and their impact on software quality. While this pattern can be observed by a static analysis on a standalone system referencing its internal functions, in a distributed system there is no possibility to get a complete view of the system through a static analysis. In the next chapter we focus on imperfect flow hierarchies, and the capability of the flow hierarchy metric to represent these anti-pattern of software engineering.

3.2.2 Measuring Imperfect Flow Hierarchies

In [Luo 2011], authors provide a study specifically focused on the flow hierarchies. Their approach has not been to focus on perfect flow hierarchies, indeed, authors state that flow hierarchies usually do not appear in a pure form in complex self-organizing systems. Instead, they focus on determining at which extent a network follows a hierarchical pattern. Indeed, some self-organizing networks are not purely hierarchical but still have some degree of hierarchy.

Self-organizing networks are usually networks which have no architects, some examples provided by authors are food webs, industrial production networks or Open Source Software. Still, these networks can partially exhibit a hierarchical structure while not having clearly identified hierarchy levels. Figure 3.1 is the graph representation of a distributed application whose vertices represent entities taking part in the system and edges, labelled *CONN* that represents the network connections between these entities. This graph is obtained by simulation of an application and is similar to the network communications graphs generated by the encoding of graphs with the model provided in previous chapter.

This graph does not match the definition of a *perfect flow hierarchy*. Indeed, some cycles can be observed, but still we can find that edges tend to have an overall global direction. Figure 3.2 focuses on two portions of the graphs, one exhibiting a strong hierarchical pattern with clearly identifiable levels, and the other one exhibiting multiple cycles and no clear levels grouping the vertices. Some edge patterns are highlighted in the second portion of the graph to exhibit the cycles and lack of hierarchical structure between some components.

To be able to detect and quantify at which extent a graph follows a flow hierarchical pattern, authors introduce a metric called the *flow hierarchy metric* (noted h). Its role is to detect and measures the extent to which locals flows (i.e. edges of the graph) follow a holistic overall underlying direction. This measure is calculated as the percentage of links that preserve the direction of the network, which translates to the number of links that are **not** involved in any cycles. Equation 3.1 represents the formula:

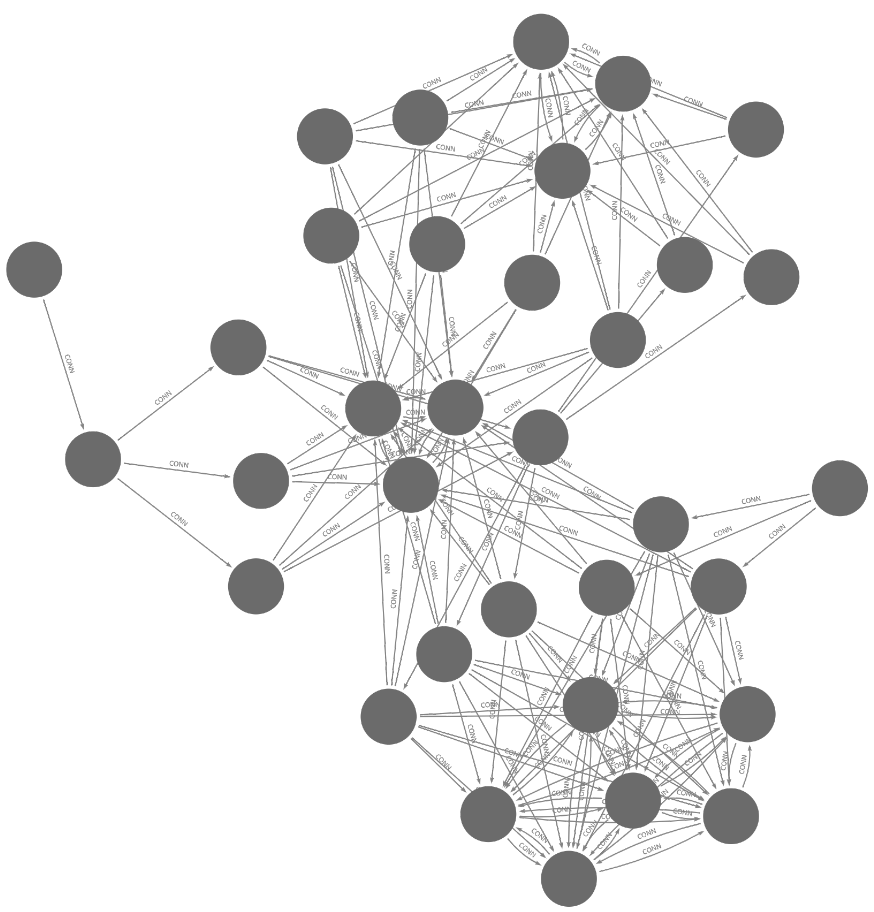

Figure 3.1: Graph Representing Communications Between Components of a Distributed Application

$$h = \frac{\sum_{i=1}^{L} e_i}{L} \tag{3.1}$$

where $L = |E|$ corresponds to the number of edges in the graph and $e_i = 0$ if the ith edge belongs to a cycle or $e_i = 1$ otherwise. Furthermore, authors also decline the formula for the calculation of the flow hierarchy metric adapted to weighted graphs. Equation 3.2 shows the adaptation for the calculation of the *weighted flow hierarchy metric* h_w:

$$h_w = \frac{\sum_{i=1}^{L} w_i e_i}{\sum_{i=1}^{L} w_i} \tag{3.2}$$

In that formula, we have still $L = |E|$ and e_i being a coefficient that identifies whether an edge belongs to a cycle or not. This calculation only considers the weights of the edges, which are noted w_i where i designates the ith edge in the graph.

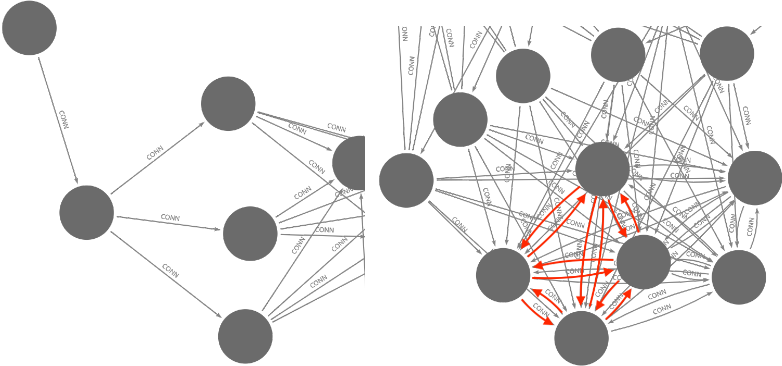

Figure 3.2: Focus on Portions of the Graph: With Clearly Identifiable Hierarchy Levels (left) and With No Identifiable Order of Vertices (right)

In its unweighted form (equation 3.1), the flow hierarchy metric corresponds to the percentage of edges contributing to retain the overall direction of the graph. Also, it can be seen as the number of edges not involved in a cycle. Equation 3.2 extends the previous definition by considering weights on the edges, this variation can be summed up as the percentages of weights involved in cycles. Still, both of these formulas do not cover the cycle detection in the graph, which is a prerequisite to get the flow hierarchy metric. They refer to the e_i coefficient associated with each edge. The next section focuses on cycle identification to associate with each of the edges the appropriate e_i coefficient.

3.2.3 Cycle Identification

In the original work, cycle identification is achieved through the exponentiation on the link adjacency matrix. In the following we consider a non-attributed graph $G = (V, E)$, we provide an overview of the method for cycle identification provided in [Luo 2011].

Figure 3.3 provides a visual representation of the computation of the link-distance matrix used by authors to identify whether the edges belong to a cycle or not. The first step is to compute the equivalent link network from the original graph, then compute its adjacency matrix. This matrix is, therefore, a square matrix of size $|E|$. By raising this matrix to the power p we obtain distance between nodes of length p. By iterating over the possible values of p we build the link-distance matrix noted M^d. If the ith coefficient on the diagonal $M^d_{(i,i)} = p$ of the link-distance matrix is not empty, therefore the edge i belongs of a cycle of length p, and therefore $e_i = 0$. Therefore, this algorithm has a complexity of $\mathcal{O}(|E|^{|V|})$.

While, the method proposed by authors allows identifying cycles, it comes with a high complexity. In addition, by computing the exponentiation of the adjacency matrix, the formula also provides the length of the cycle, a parameter that is not used in later steps of the flow hierarchy calculation.

Still, solely for the purpose of identifying cycles, some algorithms have been proved to be more efficient, in particular when we focus on Strongly Connected

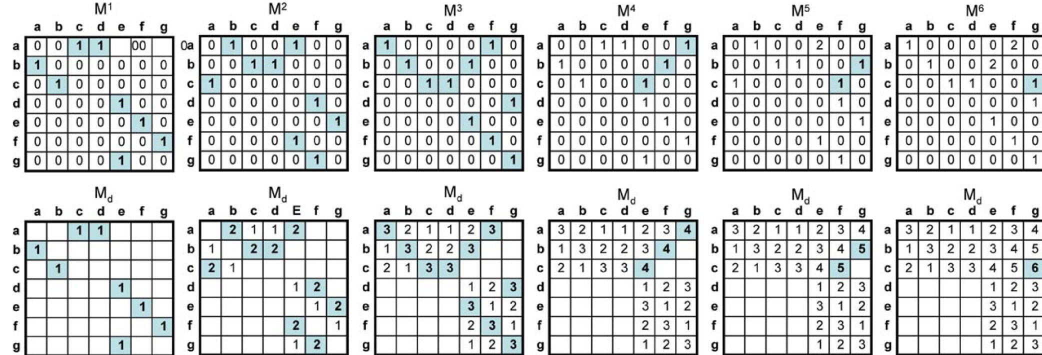

Figure 3.3: Illustration of M_d matrix computation from [Luo 2011] where M^n is the link adjacency matrix raised to the power n

Components (SCC). In a directed graph, we call a SCC a subset of the graph's vertices where there is a path between each pair of these vertices. Therefore, nodes being in the same SCC belong to a cycle. So, edges take part in a cycle when its two end vertices belong to the same SCC. As an example, figure 3.4 presents on a sample graph the SCCs in dashed lines. All edges of the graph contained in the same SCC belongs to a cycles; the ones crossing the boundaries of the SCC do not.

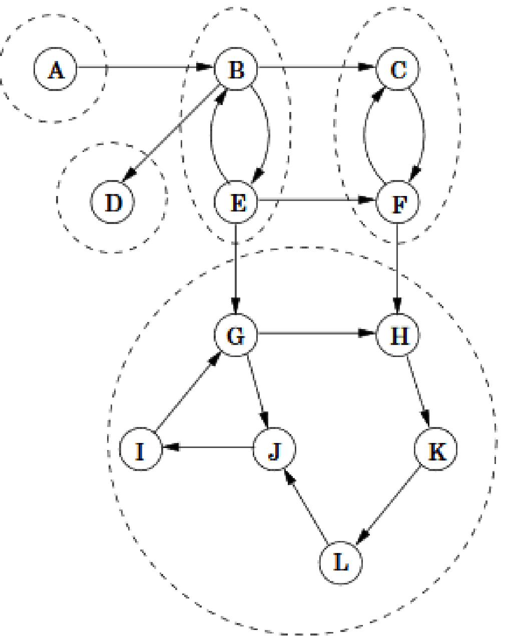

Figure 3.4: Example of Vertices Grouped by SCC in a Sample Directed Graph

Identification of SCC in a graph is now a well-known problem. In [Tarjan 1972] authors provide an algorithm capable of identifying SCC with a complexity of

$\mathcal{O}(|E|+|V|)$. To identify cycles in our flow hierarchy metric calculation, the Tarjan algorithm was chosen. It allows to identify nodes which take part in a cycle. Finally, for each edge, we create the coefficient e_i and set it to 0 if the two ends of the edge are associated with the same connected component, otherwise it is set to 1.

In this section, we provided a background study on hierarchical patterns, and set a particular focus on the one encountered in software systems. Entities communicating together forming a distributed application have been shown to follow imperfect flow hierarchies. The metric provided by [Luo 2011] can be used to demonstrate if a distributed application follows a hierarchical pattern. Still, its calculation method can be simplified by relying on more efficient cycle-identification algorithms. On the next section, we set a focus on the use of this metric on the graphs created by our trace-encoding model.

3.3 Detecting Inefficient Service Composition in Cloud Applications

Whereas Cloud Applications are effectively designed by software architects, these architects focus on the logical structure of components but not on their location. Indeed, the physical resource allocation is managed by the CSP and specifying the exact location of a service on a Cloud platform is considered to be an anti-pattern. This is the role of cloud orchestrators like *OpenStack* or *Kubernetes* to allocate VMs or containers on the underlying machines.

This resource allocation mechanism often results in inefficient placements, in particular when considering the runtime composition of resources. Also, in a performance study of the GMail Application [Ardelean 2018], authors demonstrated that user requests are far from being independent from each other. Through the user's load of the application, the communications between services can tremendously change in shape with time. Other parameters influencing the structure of the application can be listed, such as the automated tasks, the developers pushing new versions to the platform or even the Cloud Service Provider (CSP) management of resources. All these parameters influencing the network make microservices applications behave like a partially autonomous self-organizing network.

With traces, it is possible to build graphs of network communications between entities involved in a distributed application. These graphs are directed, and, as shown in the previous section, they can eventually exhibit cycles. In their initial form, traces are a DAG of *Spans*, but, when deriving the references observed over the resources layers, cycles can appear. In this section we propose to use the flow hierarchy metric to detect when the networking graphs fallback on themselves.

3.3.1 Application to the Hierarchical Property Graph

With the trace encoding method proposed in chapter 2 we are able to create graphs of the communications observed between the entities involved in serving the request. When encoding each trace with its own hierarchical property graph, we observe that, depending on the placement of micro-services, each resource commu-

nication graph can eventually exhibit cycles. Considering a Cloud-Native Application, having cycles within the topology for a single trace, spots unnecessary network calls. Networking cycles reveals an inefficient placement of the underlying resources involved in the cycle, and can contribute to performance degradation.

To illustrate a use case where the calculation of the flow hierarchy metric brings valuable feedback regarding the placement of resources, we consider the case of standard Kubernetes Cluster. With Kubernetes scaling mechanisms, when a single instance of a Pod is not enough, other instances of this pods are scheduled on other machines. Once they are ready, Kubernetes starts to send them network traffic. Figure 3.5 represents the graph encoding and rewriting process for two traces representing the same service composition, but using different resources served by a load-balancer.

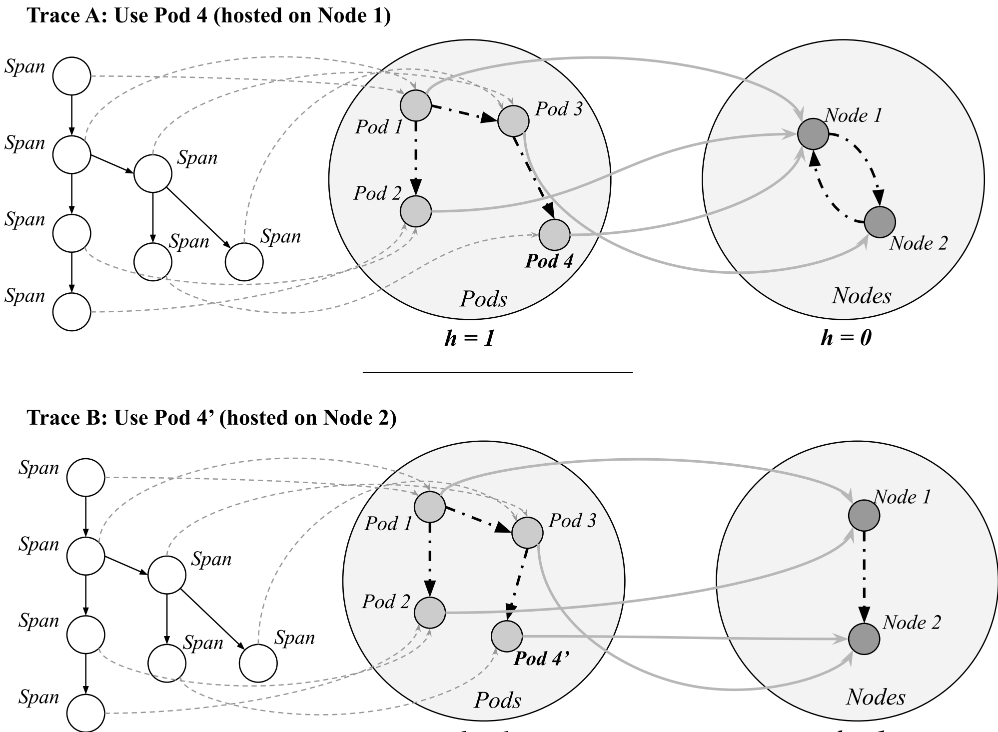

Figure 3.5: Examples of Flow Hierarchy Metric Calculation at Each Layer of the Containment Hierarchy for Two Traces

In this example, we consider a two-nodes Kubernetes cluster executing four micro-services, each of them being materialized by a Pod. *Pod 4* has been replicated twice: trace *A* shows that *Pod 4* is hosted by *Node 1*, and, on trace *B*, *Pod 4'* is hosted on *Node 2*. This example highlights that, in the case of a load balancing scenario, some service allocation may be more efficient than others. Indeed, the service composition showed by trace *B* favours more local communications within Kubernetes Nodes and favours local communications instead of distant ones.

Also, the rewriting process can lead to a projected graph made of a single vertex with no edges, e.g. when all network communications remains contained within the

containers hosted on the same node. In that case $L = |E| = 0$ and then, according to the mathematical definition, h is undefined. However, for the purpose of our model, this use case materializes a normal case where all resources are co-located under the same resource, and, therefore, network communications are efficient. As a result, when the resource graph is only made of one vertex and no edges, we define $h = 1$.

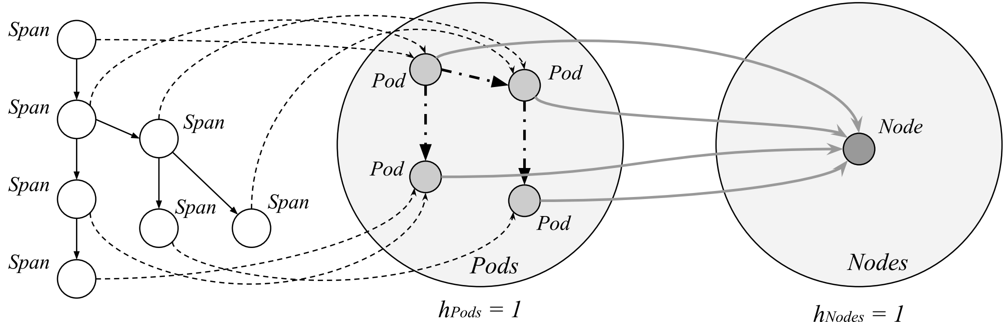

Figure 3.6: Example Where All Pods Are Executed on the Same Node

Figure 3.6 takes the previous example with all pods executed on the same node. The final graph of *Nodes* communication does not have edges, but still characterize a normal Kubernetes allocation use case. This case is considered normal and has been integrated in the formula as the intent of the flow hierarchy metric is to assess service composition.

3.3.2 Proof of Work on a Sample Cloud Application

In this section we consider an existing open source cloud application publicly available on GitHub[1] which emulate the behaviour of an online boutique. This application has been designed by Google Cloud engineers to have a demonstration application for their GCE platform. It is made of ten microservices coded in different languages, they communicate through gRPC and are instrumented with OpenCensus, a precursor of OpenTelemetry. Figure 3.7 represents the architecture diagram of the application, with the name of each of the microservices involved. This application is designed after the *API gateway* pattern, where a particular service acts as a central point of aggregation of other services results. In this section we focus on a theoretical analysis of the application composition of services, still the next section will focus on a deployment of this application on a real world Cloud Platform.

In this application, users can basically do five different operations traditionally implemented in online boutiques. In the following we detail these operations and their impact on how the components involved interact to create the final result:

1. Users can **consult the catalogue of products**: this operation will trigger the execution of five services. The resulting execution graph is a star graph with the *Frontend* service as a central part.

[1] `https://github.com/GoogleCloudPlatform/microservices-demo` Sample cloud-native application with ten microservices showcasing Cloud native technologies

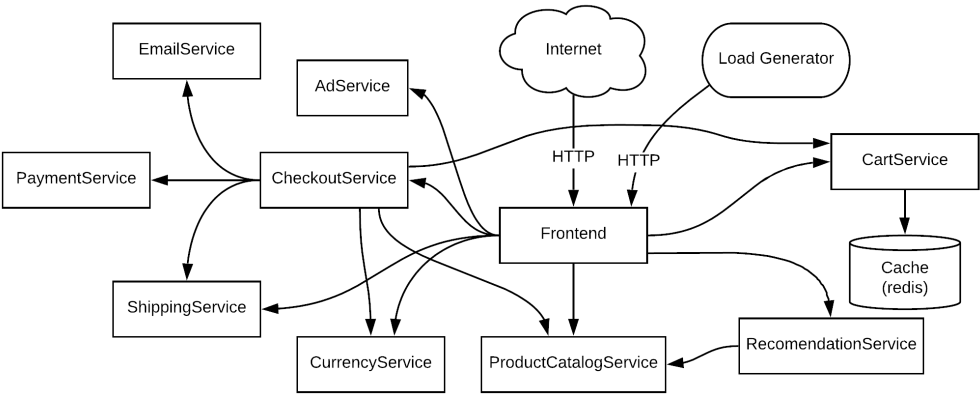

Figure 3.7: Diagram of the Microservices Demo Application with Components Interactions Provided by the documentation of the Application

2. Users can **consult the page of a specific product**: this operation will trigger the execution of six microservices. The resulting execution graph is also a star graph focused around the *Frontend* service.

3. Users can **consult their cart**: this operation also triggers the execution of six microservices and also generate a star graph around the *Frontend* service.

4. Users can **change the currency of the boutique**: this operation only triggers one microservice.

5. Users can **proceed to the checkout of the cart**: this operation involves nine out of the ten services in the application. The execution graph is a complex graph with a depth of 4.

For the purpose of detecting when the graph falls back on itself after the rewriting process, the operations detailed in the operations 1, 2, 3 and 4 are useless. Indeed, the star graphs obtained by these operations do not have sufficient depth to produce a cycle when applying the rewriting process at higher abstraction levels. Therefore, we will only focus on the *checkout* operation which provides the most complex graph of service composition.

If we suppose that our ten microservices application has been deployed on a Zonal Kubernetes Cluster made of four nodes (named $Node_1$, $Node_2$, $Node_3$ and $Node_4$) scattered in two zones (named $Zone_1$ and $Zone_2$). Nodes are distributed as the following : $Node_1 \subset Zone_1$, $Node_2 \subset Zone_1$, $Node_3 \subset Zone_2$ and $Node_4 \subset Zone_2$. Figure 3.8 represents the complete hierarchical property graphs of the composition of services required by the *checkout* operation for this particular deployment.

In this figure, on each abstraction layers, edges typed PROJECTED_REF and involved in cycles have been coloured in red. So, edges typed PROJECTED_REF in red have their associated $e_i = 0$ while the black ones have their associated $e_i = 1$.

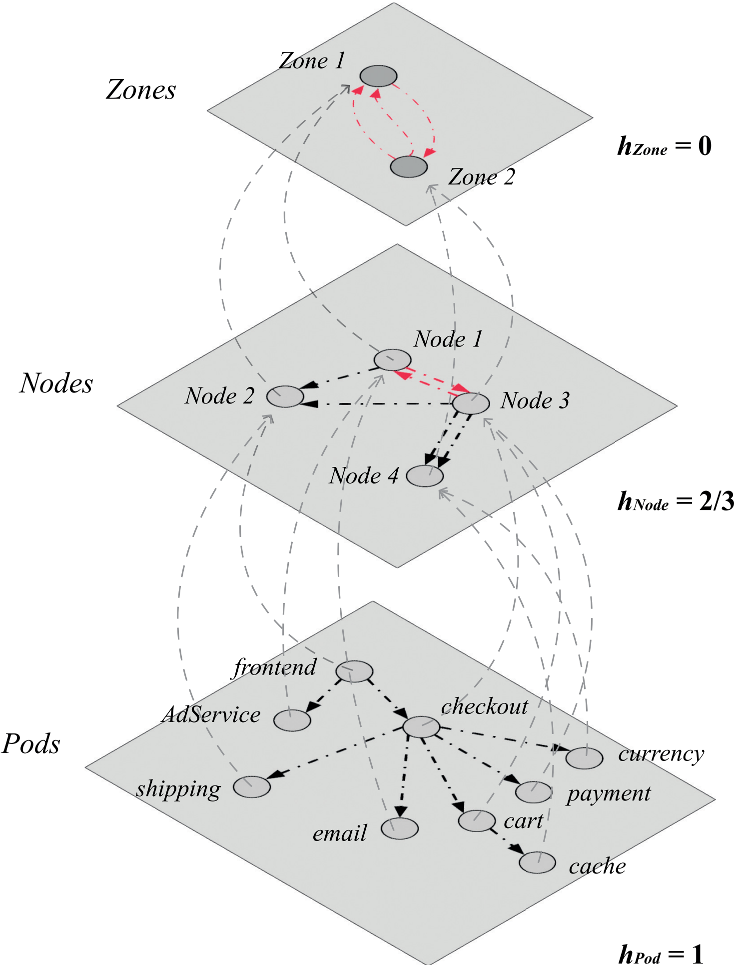

Figure 3.8: Graph Transformation for a Particular Trace

Next to each layer, the flow hierarchy metric associated to each layer has been displayed.

We can observe that an application deployed on a Kubernetes Cluster may exhibit a topology creating multiple expensive network calls. And, while traces help to understand how micro-services are composed together, they still obscure the network, which is problematic considering the current adoption of edge deployments for large-scale application. With this model, the hierarchical structure enables the analysis of the communication of a physically distributed application where the network linking elements of a distributed system is not even. The flow hierarchy metric helps to assess the composition of services by providing a numerical indicator characterizing the preservation of the structure over the abstractions levels. The next section focuses on the technical implementation of a Proof-of-Concept platform running the sample Online Boutique application presented in this section on a Zonal Kubernetes Cluster.

3.4 Implementation

To verify our approach a Proof of Concept platform has been developed to run the Online Boutique application that has been studied in the previous section. Like in the theoretical example, this application is deployed on a Zonal Kubernetes cluster. While the components of the application have not been altered, the deployment manifests have been tuned to support OpenTelemetry, which was not bundled by default. In addition, at the time of the experiments, OpenTelemetry was still under heavy development, the semantic definitions were almost stable but the collector and the agent binaries were not ready to support a production use case. Still, to illustrate the industrial use case and to mimic a real-world application, the Kubernetes deployment manifests have been tweaked to support OpenTelemetry nightly builds.

3.4.1 Designing a Multi Layers Platform with Zonal Kubernetes Cluster

Zonal Kubernetes Clusters constitute a motivating example to illustrate cases when the network linking the microservices cannot be considered even. Zonal Kubernetes in this writing refers to Multi-Zonal Clusters and sometime are also called Regional Clusters depending on the CSP own definition in its catalogue. So, behind the designation Zonal Kubernetes cluster in this thesis we consider a Kubernetes whose nodes (VM hosting the containers) are scattered in different subgroups that are linked by a particular networking link different from the networking link that links two nodes within the same group. Figure 3.9 illustrates the entities involved in normal Kubernetes clusters and in Zonal clusters. In this illustration we see that Zonal Clusters are constrained to the perimeter of a *Region*. Zonal clusters are considered to be the default solution for having a high availability cluster, still they are not a solution for scattering microservices closer to the users.

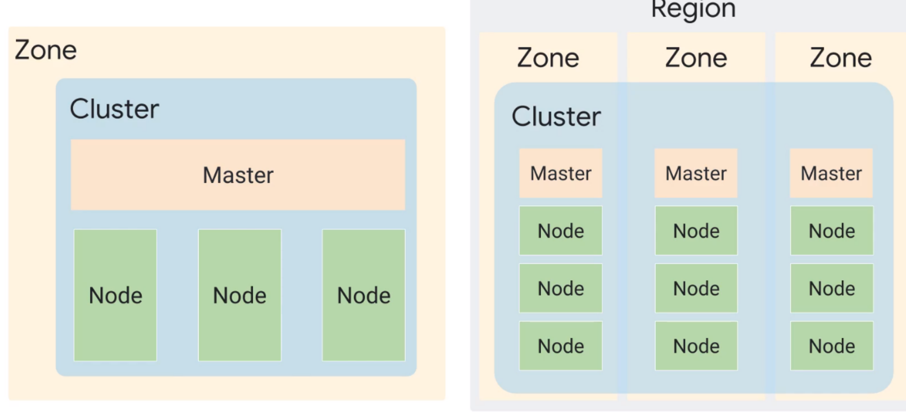

Figure 3.9: Illustration of a Normal Kubernetes Cluster (on the left) and a Zonal Kubernetes Cluster (on the right)

The limitation on scattering microservices to the edge is heavily tied to the actual implementation of Kubernetes cluster. However, several projects tightly tied to the Cloud Native Computing Foundation (CNCF) aims to remove this constraint and scatter geographically a Kubernetes application. The main projects that federate multiple Kubernetes Clusters are Gloo[2], Cilium[3] and Kubefed[4]. All of these solutions are still young and most of them were at an early prototyping stage a year ago, so they have not been considered for the experimentation part. Still, they open the room to extend the model developed for Zonal Kubernetes cluster to a federated clusters described by a containment hierarchy.

3.4.2 Getting OpenTelemetry Traces With Network Level

3.4.2.1 Instrumenting the Application

The Online Boutique sample application has been modified to support OpenTelemetry and to generate traces with network-level telemetry data. The application is made of ten microservices communicating with each other over gRPC and coded in five different languages. Some services have part of their code basis instrumented to emit traces spans based on the time spent in some functions; some have no instrumentation. This tracing implementation is made with OpenCensus, a precursor of OpenTelemetry, which is backward compatible with the Open Telemetry format used in this thesis. Also, to gain insight on the performance of non-instrumented services, application proxies have been added to each micro-service, they also emit telemetry data to complement code level instrumentation with network-level instrumentation. The tweaks realized to the deployment manifests have been to inject two additional components in each of the ten microservices:

1. An OpenTelemetry Agent which catches the Opencensus telemetry data from the application and converts them to the OpenTelemetry format. In addition, these agents enrich telemetry data with other properties defined in the OpenTelemetry semantic.

2. An application proxy that catches and forward the communications between microservices. These proxies also emit telemetry data that provides insight on the network communications between services.

Figure 3.10 shows a network interaction between two micro-services and the flow of telemetry data in the proposed implementation. Usually tracing data comes from the code of the application, however, only relying on in-app instrumentation to get traces do not provide the full picture, as it lacks networking data. In order to observe cross-services network calls and to cover the latency introduced by services,

[2]`https://www.solo.io/products/gloo-edge/` The project is defined as an API Gateway managed by a dynamic Proxy whose configuration adapt to the different clusters it links.

[3]`https://cilium.io` a Service Mesh based on the eBPF technology that provides Multi-Cluster Connectivity, in Beta at the time of writing.

[4]`https://github.com/kubernetes-sigs/kubefed` A part of the Kubernetes project in early Beta at the time of writing whose goal is to define propagation of configuration across multiple standalone clusters.

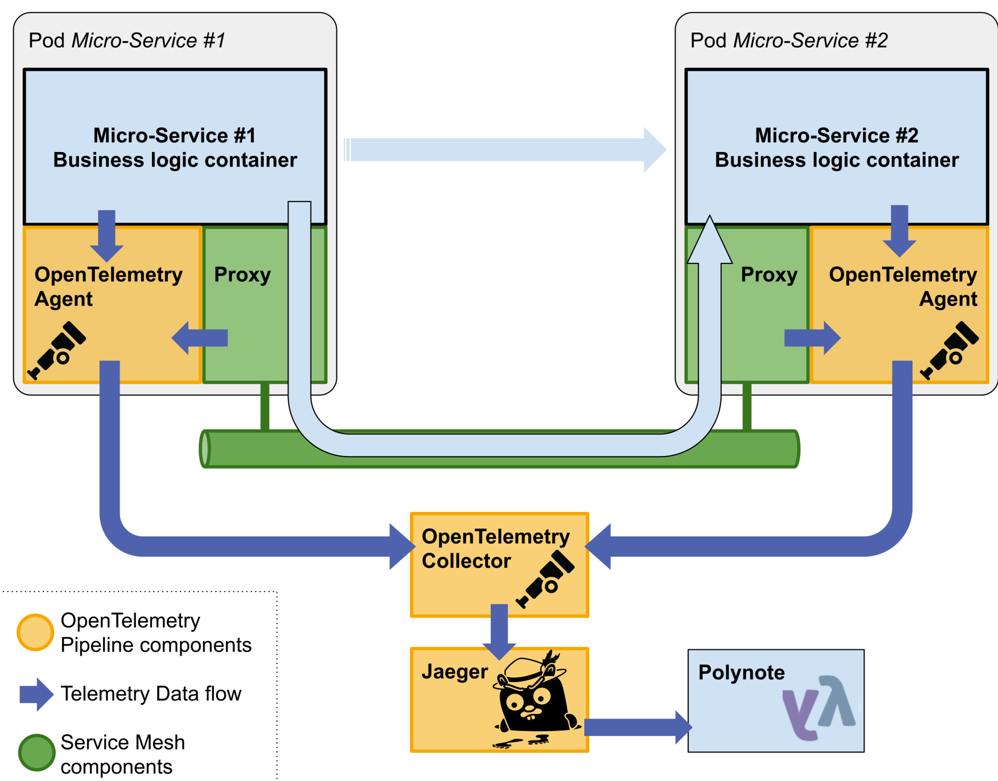

Figure 3.10: Complete Telemetry Pipeline

we extended Kubernetes with a Service-Mesh [Li 2019]. A Service-Mesh is a Data-Plane made of L4/L7 proxies injected in each microservice to better control and observe service-to-service communications. The configuration of these proxies is made through a control plane that ensures that their configuration is coherent. For the purpose of our experiment, the *Linkerd*[5] service mesh has been used as it is compatible with OpenCensus format (and thus, has a backward compatibility with current OpenTelemetry Beta binaries). With these two extra-processes added to each micro-service, we can expect clean formatted OpenTelemetry data sent to a Jaeger Tracing collector (despite Linkerd is not yet compatible with this trace format).

This implementation provides a detailed view of all the steps that takes a request of one microservice to another. Figure 3.11 set a focus on a portion of a trace illustrating the communication of the `frontend` service to the `productcatalogservice`. In this figure, each network communications in pods is represented by a red arrow numbered from 1 to 3. On the left side of the figure a trace is represented, the `frontend` pod requests the `productcatalogservice` pod, the spans highlighted in red are added by each of the proxies and reported by the OpenTelemetry agents. These proxies intercept the communications and report the response time at the boundaries of the pod. This technique also works for services which do not report

[5]https://linkerd.io/ a service mesh for Kubernetes hosted by CNCF

any spans, as its attached proxy will still report the service latency at its boundaries. As a result, pods whose code is instrumented may be perceived as a white-box but pods whose code-basis is not instrumented can still have representative tracing data, analysing its behaviours as a black-box.

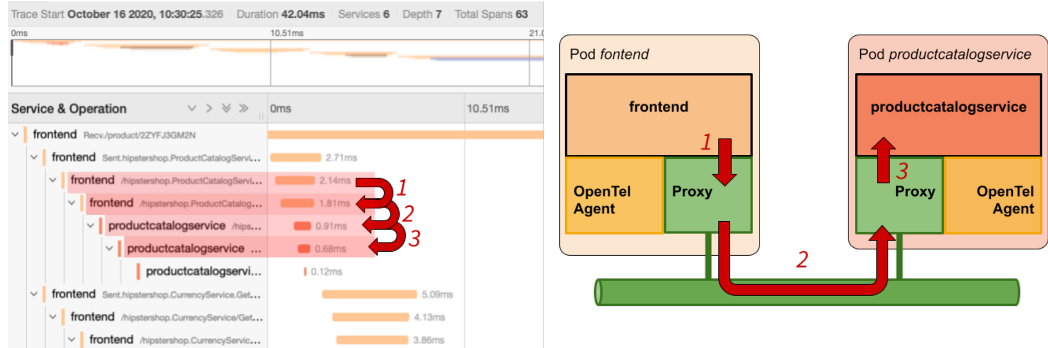

Figure 3.11: Example of the Network Communications Reported in a Trace

3.4.2.2 Adding Networking Latency to Edges Weights

These traces also highlight the time spent on the network in a dependency chain: Figure 3.11 provides the details of the networking delays induced by the tracing pipeline on a particular requests. We can observe that the `frontend` microservice requests the `productcatalogservice`: the calling function in the first microservice last for 2.71 ms and the responding function in the distant microservice only takes 0.12 ms to reply. The time spent on the network for this exchange is $2.71\ ms - 0.12\ ms = 2.59\ ms$. The additional proxies injected provide further decomposition of this latency: The arrow 1 shows that a local communication within a `frontend` pod added $2.14\ ms - 1.81\ ms = 0,33\ ms$, and the arrow 3 shows that the local communication within the `productcatalogservice` pod added $0.91\ ms - 0.68\ ms = 0.23\ ms$. Finally, the arrow 2 showed the time spent on the overlay network linking the pods which induced a latency of $1.81\ ms - 0.91\ ms = 0.9\ ms$.

This implementation normalizes the the latency measurements at the boundaries of microservices and focuses on network induced latency, which was not present in the original application tracing. It also provides a better granularity of measures but traces with this method are more dense and complex to analyse for humans. By applying the graph encoding and rewriting method from chapter 2 on figure 3.11, only the communication marked 2 is preserved.

Therefore, during the rewriting process, it is possible to extract properties from the vertices and edges of the graphs to compute new properties that will be embedded in the final graph. Edges typed `"PROJECTED_REF"` can be enriched with the computation of the network time as described in the previous paragraph. Still, no further studies have been conducted with the analysis of the impact of the network time. Indeed, at the scale of the application we cannot observe a noticeable latency increase due to cross-zones communications.

3.4.3 Computing the Flow Hierarchy Metric

The calculation of the flow hierarchy metric h was computed online for each of the incoming trace in the pipeline and was done after the rewriting process. Identification of Strongly Connected Componentss (SCCs), is based on the implementation provided by Gremlin with the method `g.V().connectedComponent()`. This function has been configured to compute SCCs instead of Connected Components. The difference is that SCC follow the direction of the edges whereas for Connected Components, two nodes belong to the same connected component if they are linked by an edge without considering its direction.

Listing 4 details the Gremlin query embedded in the Scala code after the graph rewriting part: at lines 4 and 5 the `g.V().connectedComponent()` function is refined overwriting standard behaviour thanks to the ``.`with`()`` method. By specifying, on the line 5, the custom Gremlin traversal query `__.outE("PROJECTED_REF")` we indicate to the Gremlin traversal engine to only compute components by following the direction of the edges typed `"PROJECTED_REF"`. This variable is part of a function called `stronglyConnectedComponents` but the rest of the function has been omitted as it does not bring much value and can still be found in appendix B.

```
1 val scc =
2   gJava.V()
3     .connectedComponent()
4         .`with`(ConnectedComponent.propertyName, "component")
5         .`with`(ConnectedComponent.edges, __.outE("PROJECTED_DEP"))
6     .project("id", "component")
7         .by(__.id())
8         .by("component")
9     .toList()
```

Listing 4: Computation of SCC with Gremlin: assign the variable `scc` to the list of all vertices ID with their component ID

The previous Gremlin query associates a vertex ID with a unique component ID Later steps consist of integrating this information into the vertices of the graph as properties. Once each of the vertices has its SCC ID set in the `"component"` property, computing the flow hierarchy metric is a straightforward process: Edges whose start and end vertices belong to the same component are involved in a cycle. On the contrary, edges whose start vertex has a different component ID from its end do not belong to a cycle. Therefore, h can be computed as the ratio of the number of edges not involved in a cycle by the total number of edges.

Listing 5 provides the full definition of the unweighted flow hierarchy computation function: the variable `sccIndexedEdges` is obtained by a gremlin query returning all edges with the pair of component ID of the source vertex and destination vertex of each edges. This variable is then filtered (line 19) to only obtain out-of-cycle edges. The value of h corresponds to the size of the filtered collection of edges divided by the size of the complete collection when there is at least one

edge in the graph.

```
def flowHierarchy(graph: ScalaGraph): Double = {
  import TraceMetaModel._

  val scc = stronglyConnectedComponents(graph)
  val g = scc.traversal

  val sccIndexedEdges =
    g.E.project(
      _(By(__.inV.properties("component")))
      .and(By(__.outV.properties("component")))
    )
    .toList
    .map {
      case (kvSrc, kvDst) => (kvSrc.value(), kvDst.value())
    }

  val countEdgesOutOfCycles =
    sccIndexedEdges
      .filter { case (cSrc, cDst) => cSrc != cDst }
      .size
      .toDouble

  val totalEdges = sccIndexedEdges.size.toDouble

  if (totalEdges != 0) {
    countEdgesOutOfCycles / totalWeight
  } else {
    1.0
  }
}
```

Listing 5: Computation of the flow hierarchy metric

Whereas listing 5 provides a basic implementation of the flow hierarchy metric calculation, this code has been generalized to also support the weighted formula as provided in equation 3.2. Appendix B.4 provides a more generic definition of the flow hierarchy computation function working for both weighted and unweighted computations. This function takes an additional parameters that handles the weighted flow hierarchy computation formula.

From a technological standpoint, all these computations are performed on In-Memory Graphs managed by the Gremlin library. Each traces are processed independently and can be run in parallel. The following section provides a study on the results of computing this flow hierarchy on traces.

3.4.4 Results

By utilizing the flow hierarchy metric on each layer of the model, we can address the issue reported by Orange in the Djingo project. Indeed, the flow hierarchy metric allows to identify requests which loop among data centres. Since the formulation of the problem, the project has undergone a massive structural changes and architects decided to co-locate resource in the same data centre. This decision made impossible to run a real-world experiment on the target platform. Therefore, the following section provides a study on Zonal Kubernetes Clusters which mimic the pattern of resources geographically distributed. This pattern is observed on many large-scale Cloud Applications and tends to become an industry standard for High-performance application, in particular considering use cases like multimedia streaming, or real-time applications.

3.4.4.1 Analysing Traces of the Proof-of-Concept Platform

A Zonal Kubernetes Cluster has been instantiated on the Google Cloud Platform with four Nodes scattered in two different availability zones: `europe-west1-b` and `europe-west1-c`. The Online Boutique described in section 3.3.2 has been deployed into this cluster. Each of the micro-services has been replicated four times with the constraint that two replicas of an individual service cannot be hosted on the same machine. As a result, each of the Kubernetes nodes hosted a single instance of each service. The internal load-balancing configuration has been left untouched; the balancing method used was a basic Round Robin that alternates through services. Therefore this scenario aims to describe all possible service compositions that can be encountered for a given application in a cluster.

For the experimentation, each trace was processed to generate a single trace graph exhibiting the resource containment hierarchy $Pods \subset Nodes \subset Zones$. Therefore for each trace, a tuple $(h_{Pods}, h_{Nodes}, h_{Zones})$ was generated. Figure 3.12 plots these flow hierarchy metric values depending on the total duration of the request (expressed in ns). The first graph in figure 3.12 represents an histogram of the distribution of the request based on their total duration. In the second graph, each point represents, for a given trace, the flow hierarchy metric calculated for the graph of *Pods*. The third and the fourth graph represent, the calculation of the flow hierarchy metric respectively calculated for the graphs of *Nodes* and the *Zones*.

We can observe that, for many traces, the graph of resources falls back on itself after following the containment hierarchy. For Zonal Kubernetes Clusters, these plots cannot expose a direct impact of the flow hierarchy measure on the application response time. And while, the graph demonstrates that the flow hierarchy metric tends to lower when we consider the composition of higher level resources, the presence of cycles does not seems to alter the application global latency.

3.4.4.2 Limitation and Experimentation Representativity

Whereas Zonal Kubernetes Clusters exhibit a similar architecture as the multi data centre deployment of the Djingo project, they are still built with different assumptions in mind. In Djingo, scattering services across multiple data centres was meant

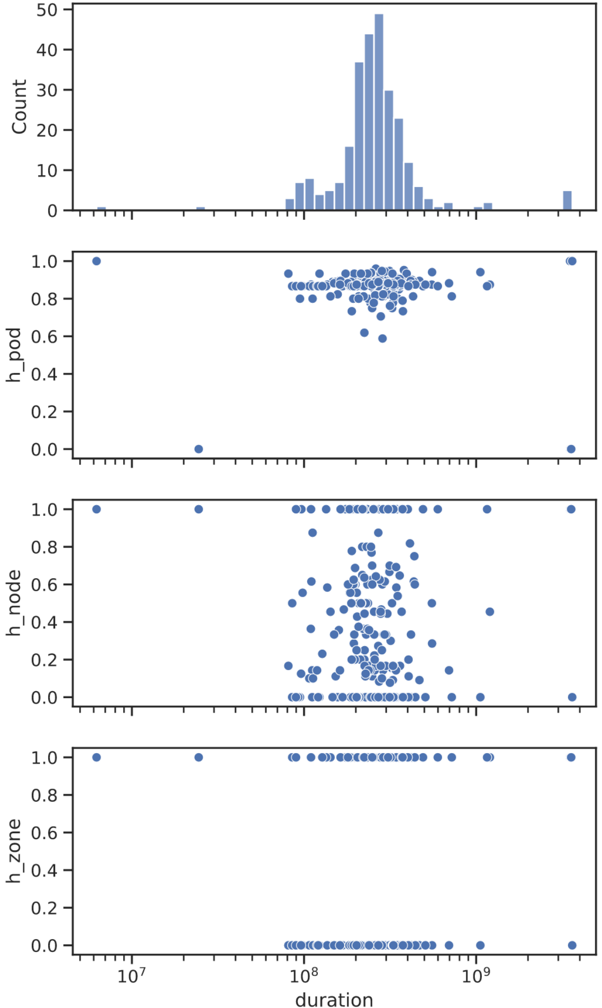

Figure 3.12: Graphical representation of traces and their flow hierarchy metric ($h_{Pods}, h_{Nodes}, h_{Zones}$) depending on the end-to-end response time of the trace

to support proximity with end-users. Whereas, the Zonal Clusters deployed in our Proof-of-Concept are designed ensure availability in case of failure while keeping the latency as low as possible. In both cases, the network linking services cannot be considered even, but our experimentation showed that this irregularity materializes in two different ways: On the one hand, the cost potentially manifests in terms of latency, whereas, on the other hand, the cost is simply monetary.

Multiple reasons can explain this behaviour:

1. The scale of the cluster can barely represent a production-ready application: the load of the service is not comparable. Indeed, Cloud application deployed in a *Development environment* and in a *production environment* do not exhibit the same performance anomalies. Performance anomalies are only observed in production environment with a real load.

2. Among the multiple causes of latency that can be met in a Cloud environment, cycles are not the predominant one on this platform. For instrumentation purposes each microservice was added two extra programs that live within each Pod.

3. The instrumentation libraries were still in Beta at the time of testing and were not suitable for a production use case. They did not meet the requirements to be deployed in an application running real user traffic.

The representativeness of the platform was bounded to all of these constraints, and therefore it did not provide an environment capable of exhibiting the desirable behaviour. Indeed, the monitoring need expressed by the Djingo operational team was to know whether or not the latency peaks observed on some requests may be caused by unnecessary communication between data centres. However, during the development of the thesis, the team developing Djingo decided to opt for a single cluster and discontinue most of the features from the original project. Still, whereas the impact of cycles on the latency has been difficult to evaluate in the case of a Zonal Kubernetes Clusters, the Proof-of-Concept has shown that it is possible to correlate the latency of a request with the presence of inefficient communications.

3.5 Conclusions

In this chapter, we leveraged the model proposed in the previous chapter to address the problematic initially formulated by Djingo teams in Orange. The concept of flow hierarchy metric has been presented to detect the presence of cycles in a graph. In Cloud applications, cycles translate in an inefficient composition of resources that can only be observed at runtime. The flow hierarchy metric, propose to sum up a trace, which is a complex data format into a numerical indicator translating the extent to which the composition of resource preserve a hierarchical structure. For multiple external reasons, the experimentation has not been able to run on real world data. Instead a Proof-of-concept platform has been designed with state-of-the-art Cloud technologies to match a problem of identifying the impact of cycles in the composition of resources in a distributed application.

The work presented in this chapter, along with the description of the model presented in Chapter 2, has been published in [Cassé 2021].

Regarding the Proof-of-Concept platform, the technological decisions to rely on Service Meshes was inline with the strategy of Orange to study the potential of this technology. Through this work, motivating examples have been provided to highlight their capabilities to better control and observe the network linking micro-services. Linkerd was one of the most mature Service-Mesh implementation along with Istio[6], and for the purpose of this thesis Linkerd was picked because of its compatibility with OpenTelemetry. With the recent advances in technology today, Cilium[7] now appears to be a more suitable choice as it does not relies on adding

[6] `https://istio.io`
[7] `https://cilium.io`

containers to each pods in Kubernetes. On the contrary, Cilium relies on instrumenting the linux kernel with eBPF to modify the overlay network configuration at runtime. In addition, Cilium also provides support for distributed Kubernetes Clusters while reporting its metrics in the OpenTelemetry format. This technology appears to be a promising candidate whose instrumentation is not as invasive as the one picked for the experiments described in this chapter, without changing its format.

Chapter 4

Identifying Bottlenecks with Graph Centrality Analysis

Contents

4.1 Introduction 81

4.2 Generalizing the Graph Encoding Model 83

4.2.1 Including Multiple Resource Type in the Model 83

4.2.2 Configuring the Containment Hierarchy 85

4.2.3 Characterizing an AWS Application 87

4.3 Application to Complex Cloud Applications 88

4.3.1 Overview of Graph Centrality Algorithm 89

4.3.2 Distributed Applications Bottlenecks 92

4.4 Implementation 93

4.4.1 Using Spigo for Emitting OpenTelemetry Traces 94

4.4.2 Scenario Selection and Representativeness 95

4.4.3 Observing the Impact of Betweenness Centrality in the Riak Simulation 97

4.5 Conclusions 101

4.1 Introduction

In this chapter, we propose to leverage the model to exhibit another anomalous pattern that is commonly encountered in traditional Cloud Applications. Instead of analysing each trace independently, we propose an approach that considers the application as a whole. This approach uses the global communication graph of resources managed by the graph encoding pipeline presented in figure 2.13 on page 51. Indeed, the provided model is capable of maintaining a hierarchical graph representing the communication of resources from multiple abstraction layers.

The scale of the application used in the previous chapter for the experiment is still small compared to the real-world example encountered in Orange. And, whereas the previous application was considered to be small, running on a Zonal Kubernetes Cluster these ten microservices still had an important cost. This section is more focused on larger scale applications that cannot be easily grasped by a human operator. For this reason a simulation environment is proposed to create a large scale complex *Death-Star* Application.

In this chapter, we propose to address a challenge commonly encountered in distributed system monitoring: bottleneck identification. The layered graph model constitutes an opportunity of tackle down bottleneck identification under a different angle. The proposed model is capable of maintaining a high level view of a distributed application in the form of a property graph. By running centrality algorithm on this model, the most important vertices can be highlighted. Whereas the definition of importance may vary depending of the algorithm, in the following chapter, a theoretical study is provided on graph centrality algorithms. Then we apply these definitions to the context of cross components communication in a distributed Cloud application.

To support this study, experiments have been made in simulation environment that has been tuned to emit OpenTelemetry traces. This simulation environment follows the model of AWS-deployed Web Applications, it is capable of emulating a geographically distributed application. By analysing traces returned by this simulation, we demonstrate the generic approach of the model. Whereas the simulation environment can simulate multiple zones and regions in AWS, the model is not strictly the same as the one described in Zonal Kubernetes Cluster covered in chapter 2. This chapter therefore details the approach that has been taken to make the containment hierarchy a configurable parameter of the model.

Finally, the results of the simulation are presented: the simulated application was designed to rely on the geographical distribution of computation units. We considered the case of scaling the number of instance of a particular service that, after an architectural analysis, was acting as a bottleneck within the data ingestion pipeline of the application. By changing the number of replicas of this bottleneck service, we observed the evolution of the centrality score of these services. Therefore, this chapter provides a study on identifying the bottleneck services in a Cloud Application by running centrality algorithm on the graph maintained by traces.

This chapter is structured as follow:

Section 4.2 extends the work presented in Chapter 2 by providing an extension to the initial model. In this use case, the approach of identifying resources of a Zonal Kubernetes Clusters has been generalized to match Most of OpenTelemetry Resources types defined in the semantic specification. The order of containment of resource type is defined as a parameter of the model. The first part of this section is focused on bringing generality to the approach defined earlier. Meanwhile, in the second part, the focus is set on the application of this model to AWS structured Applications.

Section 4.3 presents the graph centrality algorithms and their uses on the model hierarchical model. In particular, this section details the betweenness centrality measurement and its meaning in the context of our Cloud Application Model.

Section 4.4 presents the simulation environment: Both the simulation software and the choice of the scenario are detailed. Through this simulation, the number of instances of a critical service is changed and we observe the evolution of ranking returned by the betweenness centrality algorithm.

4.2 Generalizing the Graph Encoding Model

The model presented in chapter 2 aims to describe geographically distributed applications. At this point of the document, its use case has only been materialized on a Zonal Kubernetes Clusters. However, these Zonal Clusters are not the only Cloud technology that supports a geographical distribution of resources. Indeed, in [Gonigberg 2018], a publication on Netflix Technological Blog, authors describe their application proxy named *Zuul* which leverages the notion of "availability zones" and "regions" provided by AWS to scatter an application over the globe while maintaining high availability SLAs. In this work, and more generally in the immense majority of physically distributed Cloud providers, the computing resources executing the application also obey to a containment hierarchy.

Also, the OpenTelemetry semantic for describing these computing resources is designed to express these concepts of availability zones, regions too. Therefore, traces also have the potential of expressing any embedding of resources in the same way they expressed the Zonal Kubernetes Clusters containment logic. Table 2.1 from page 36 presents the attributes `cloud.region` and `cloud.availability_zone` which characterize the physical location of the resources. And, while OpenTelemetry adoption is driven by cloud technologies, and mainly Kubernetes, it remains an open project that has the potential to extend to new use cases.

In the following, an approach for generalizing the model is provided: the graph encoding method presented earlier is decomposed in two different steps to extract multiple resources from the metadata, and also to configure the containment hierarchy.

4.2.1 Including Multiple Resource Type in the Model

The OpenTelemetry resource semantic is still improving and aims to cover more and more concepts, not only related to traditional Kubernetes Clusters. Topics like "Function-as-a-Service" or "end-device performance measurement" are now part of the experimental specification. These topics take place in the context of providing better insight in tracing data and to have the relevant metadata for characterizing resources. For the *Djingo* Application multiple data centre were used, both to ensure reliability but also to be closer to users. Scattering resources in multiple data centre eventually from multiple CSPs has multiple business usages that resonate with active research topics.

Therefore, while the main idea behind the model presented earlier remains the same, the graph encoding method has been altered to identify the resource type defined in the OpenTelemetry semantic and to be more flexible. The graph encoding method has been turned into a two steps process where, in a first step the metadata present in the resource description is translated into vertices, and in a second step the containment hierarchy is added to the graph. Table 4.1 presents visually the matching that is achieved from encoding metadata attributes to a property graph. This encoding techniques allows to create custom labels for vertices which are written in a capsule in the table. Also the vertex properties are listed in the top left corner above the circle.

Resource Attributes	Graph Vertices
`k8s.pod.name` and `k8s.pod.uid`. Also `k8s.pod.ip` was used for IP address but it was removed from the standard.	uid: name: Pod
`k8s.node.name` and `k8s.node.uid`. Also `host.ip` was used for IP address but it was removed from the standard.	uid: name: Node
`k8s.cluster.name`	name: Cluster
`cloud.availability_zone` or `cloud.zone` depending on the version of the standard.	name: Zone
`cloud.region`	name: Region
`service.name`, `service.namespace` and `service.instance.id` define a service **instance**.	name: namespace: instance.id: Service

Table 4.1: OpenTelemetry Cloud Semantic for Graph Encoding

This table is not exhaustive. Indeed, in its last version, the semantic covers more that 80 different attributes for defining resources. The table only extends the resources presented with the one that will be used in the following for analysing simulation results. Therefore, we expanded the encoding logic to support the identification of *regions* and *service instances*. The *service instances* vertices have been simply labeled *service* to have more concise illustrations. A vertex labeled *Service* in the following characterize only one instance of a particular service when it is replicated and part of a load-balancing group.

As a result, the first step of the graph encoding generates multiple vertices based on the resource attributes in the trace but does not express the containment hierarchy. Figure 4.1 represents the meta model of the new graph where resources can be matched from the attributes presented in table 4.1. For examples, traces captured in a AWS application not relying on Kubernetes will not have all the attributes starting with `k8s.`. Still the attributes `cloud.region` and `cloud.zone` and also `service.name` and `service.instance.id` will be likely to be present. Therefore, a normal AWS application will created traces that will result in vertices labeled *Service*, *Zone* and *Region*. If, during the development, the application developers subscribe to EKS (the Kubernetes cluster managed by AWS), the traces will have these new attributes and the resources will be encoded in the graph.

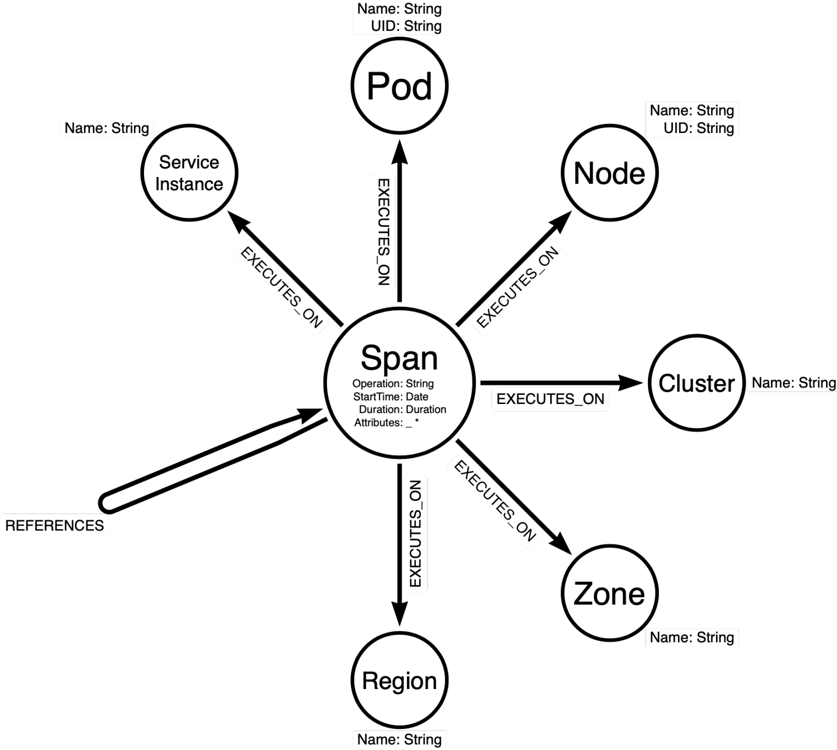

Figure 4.1: Resource Base-Graph for a Generic Meta Model

Still, adding a Kubernetes abstraction in an application will change the embedding of resources, and in particular their order of containment. The next section discusses how the containment hierarchy is added.

4.2.2 Configuring the Containment Hierarchy

The relationships typed `IS_CONTAINED` are a fundamental building bloc of the model and of the graph rewriting technique; they allow the creation of a hierarchical model. The multiplication of resources presented in previous section raised questions on the generality of the approach. Not all relationship `IS_CONTAINED` can be hard-coded in the graph model, instead, the order of embedding of resources should be a parameter of the model. Indeed, in practice, when building a Cloud Application, the architectural changes that disturb the containment hierarchy order, are extremely rare events that are planned by operational teams.

In order to add the relevant `IS_CONTAINED` relationships, a graph rewriting approach has also been used: the embedding of resources materialized by the notation $Pod \subset Node \subset Zone \subset Region$ is converted in a pattern that matches resources labels and creates the missing edges. Equation 4.1 provides the pattern that identifies all tuples of vertices (r_1, r_2) where r_1 is a *Pod* and r_2 a *Node*. With graph rewriting, these two vertices linked to the same *Span* will be linked by an `IS_CONTAINED` relationship. The first line of this equation designates the vertex pattern $\chi_1 = (r_1, Pod, \varnothing)$ that identifies a vertex labeled *Pod* and name it r_1 for

future usage. The last line designates the vertex pattern χ_3 that identifies a vertex labeled *Zone*. Finally, between these two vertex patterns, there is the pattern of the path that separates these two vertices: a *Span* labeled vertex and two outgoing edges pointing to the r_1 and r_2 resources.

$$\begin{aligned}\pi = &(r_1, Pod, \varnothing),\\ &(\leftarrow, \text{nil}, \text{EXECUTES_ON}, \varnothing, (1,1)),\\ &(\text{nil}, \text{Span}, \varnothing),\\ &(\rightarrow, \text{nil}, \text{EXECUTES_ON}, \varnothing, (1,1)),\\ &(r_2, Zone, \varnothing)\end{aligned} \tag{4.1}$$

Therefore, by applying this pattern at each containment level, all tuple of vertices where an edges typed `IS_CONTAINED` can be identified in the graph encoding the traces. After the application of the rewriting pattern, each trace graph exposes the embedding of resources. Figure 4.2 represents the result of graph rewriting process on the Meta-Model: the initial Meta-model from figure 4.1 is presented on the left. On the right side, the graph represents the final meta model for the resource embedding $Pod \subset Node \subset Zone \subset Region$.

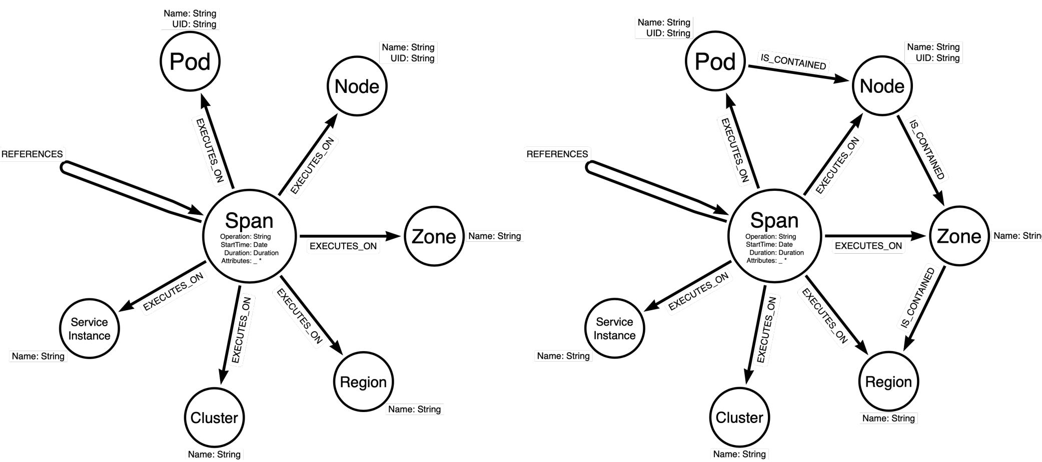

Figure 4.2: Example of adding the containment of resources for the embedding $Pod \subset Node \subset Zone \subset Region$.

To support a wider variety of Cloud application architectures, the graph encoding process initially presented in chapter 2 has been tuned to support multiple types of resource and also being parameterized. With this approach, the concept of resource embedding for Zonal Kubernetes Cluster can be supported by all hierarchical applications whose layers are materialized by OpenTelemetry attributes. Still, to expose the relevant abstraction layers, the resource embedding needs to be formalized as an input for the graph encoding technique. The next section provides an example of another type of geographically distributed application: Amazon Web

Services Application often leverage all the data centres to be the closest to the user and have a low latency.

4.2.3 Characterizing an AWS Application

Amazon Web Services Applications are notorious examples of applications leveraging the geographical distribution of microservices among data centres. The streaming platforms *Netflix* and *Twitch*, both based on AWS built an application minimizing latency by using the various data centre of the provider as point of presence to have the lowest possible latency [Gonigberg 2018, Böttger 2018, Deng 2017].

These computing resources also follow a containment hierarchy that is defined in the OpenTelemetry Resource Semantic. In global AWS applications, each microservice belongs an *Availability Zone*, and each of them belongs to a *Region*. Unlike for *Zonal Kubernetes Clusters*, it is common for applications to be deployed in multiple zones. All these layers have been presented in table 4.1 and can be encoded by the graph model parameterized with the containment relationship: $Service \subset Zone \subset Region$. Figure 4.3 represents this graph encoding meta-model.

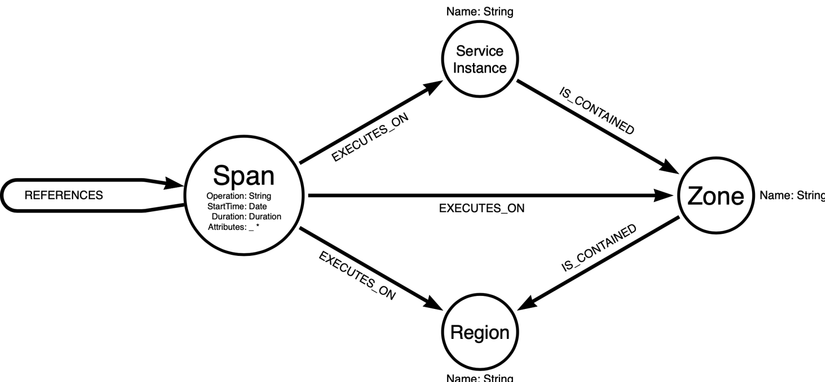

Figure 4.3: Meta Model of the graph encoding applied to traces for AWS Applications

Netflix, in particular, has published multiple engineering works on creating a load-balancer capable of considering these geographical constraints. This work materialises as an API-Gateway[1] named *Zuul* and published in open source[2]. While the following work takes these Zuul Applications as an example, deploying and evaluating the performance of an application at this scale would require a tremendous amount of money. Indeed, maintaining the Zonal Kubernetes Cluster from previous work already lead to bill climbing from one to three hundreds of euros per months, the scale of such an application would be orders of magnituides higher.

For this reasons, the *spigo*[3] simulation program will be used in the later. *Spigo* author is Adrian Cockcroft, a former Netflix Engineer that created this simulation

[1] A microservice playing the role of routing the user requests of a Cloud application to the right instance of the microservice to process it

[2] `https://github.com/Netflix/zuul` Public repository of the Zuul API gateway

[3] `https://github.com/adrianco/spigo` Public repository of the spigo simulation program

environment to model the component interactions in a global AWS Application. This is the application that was used to create the so-called *Death Stars* from figure 1.2 presented in page 21.

Therefore, by decomposing the trace-to-graph encoding process in two steps, we have been able to describe a wider variety of Cloud-based architectures. Throughout this chapter we will consider these Amazon web services applications replicated in multiple data-centre in the world. For an application of this scale the graph of the communication of services will provide a new insight on the overall application performance. In the next section we focus on a theoretical study on graph centrality analysis, then we propose to evaluate the impact of some centrality measures in a simulated Cloud application.

4.3 Application to Complex Cloud Applications

In chapter 2, a full graph encoding pipeline was proposed: each trace is encoded at runtime, then the trace graph was added to a hierarchical graph that accumulates all the traces and is stored in Neo4j. While the work presented on flow hierarchy was executed online for each traces, in this chapter we focus on the analysis of the knowledge graph. Figure 4.4 recalls the pipeline presented in section 2.5, the final stage of the pipeline is the merging of the trace graph within the knowledge graph.

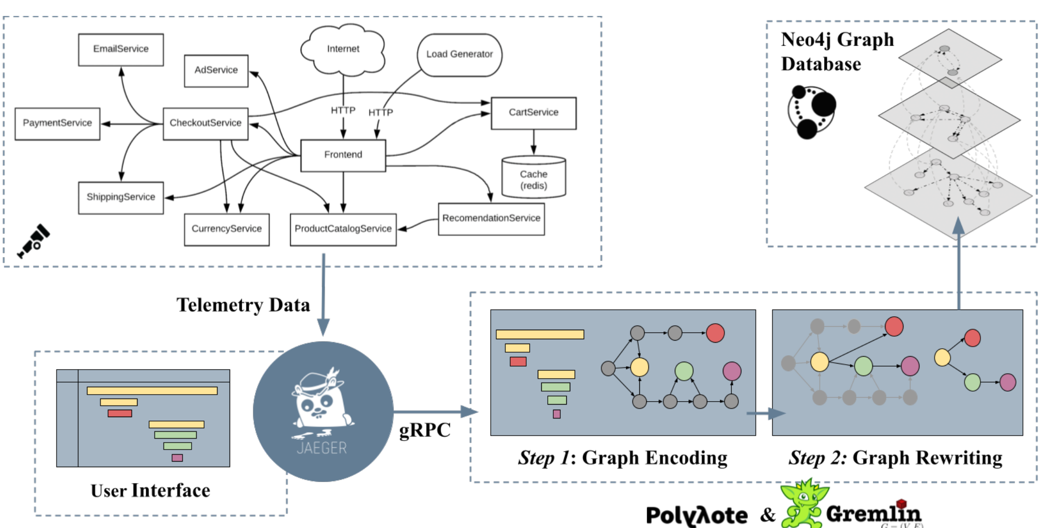

Figure 4.4: OpenTelemetry graph encoding pipeline

The proposed model and techniques leverage the heavily connected nature of traces and their semantic to maintain a hierarchical property graph. Each layer of the hierarchical graph is, itself, a directed graph whose vertices model the resources and whose edges materialize their communications. As a result, each graph may express different types of topology, having their own characteristics. The main goal of having a global network-centric model in a distributed application is to extract

a global view of the application behaviour.

Microservice applications quickly grow in size, and performance anomalies occurring in this wide crowd of services may not have the same effect on the overall application performance. For this reason we propose a technique that highlights critical services based on their communications within a global application. One of the most prominent challenges when monitoring such an application is to identify bottlenecks in the system. In the following we present centrality algorithms and their application for the graph maintained by the trace encoding pipeline.

4.3.1 Overview of Graph Centrality Algorithm

Centrality analysis designates the fact of identifying the most **important** vertices in a graph. This definition of importance may vary depending on the use case, and therefore multiple algorithms exists. In the following, a review of the most common graph centrality algorithm is proposed: a focus is made on the definition of importance these algorithms express. Then, in a later section, these definitions are set on perspective for graphs exhibiting network communications in a distributed environment. Still, graph centrality analysis remains a vast and well studied problem [Freeman 1979]. It has witnessed a wide variety of applications: whether being for general anomaly detection in graphs [Akoglu 2015] or for studying communication networks, sociology, geography or even protein network analysis [Klein 2010, Jeong 2001].

To better define the computation of centrality scores in a graph, the following concepts are defined to support a later formal definition of centrality computations. The following supposes a graph is noted $G = (V, E)$ where V is the set of vertices and E the set of edges:

- **Degree**: the degree of a vertex $v \in V$ is denoted k_v, it represents the total number of neighbours of the vertex v.
- **In-Degree**: the in-degree of a vertex $v \in V$ is denoted k_v^{in}; for directed graphs, the in-degree of a vertex v materializes the total number of its neighbour that have a direct edge toward that vertex.
- **Out-Degree**: the out-degree of a vertex $v \in V$ is denoted k_v^{out}; similarly to the in-degree, the out-degree of a vertex v materializes the total number of its neighbours that can be reached via an edge directed from the vertex v toward them.
- **Neighbour Set**: The neighbour set is denoted $M(v)$, it is a set containing all the vertices adjacent to $v \in V$.
- **Distance**: the distance between two vertices $v_1, v_2 \in V$ is denoted $d(v_1, v_2)$. The distance represents, if it exists, the length of the shortest path between v_1 and v_2.
- We also propose the notations σ_{st} and $\sigma_{st}(v)$ to represent, respectively, the number of shortest path between the vertices s and t, and the number of

shortest paths between s and t where the v acts as an intermediate vertex in the path.

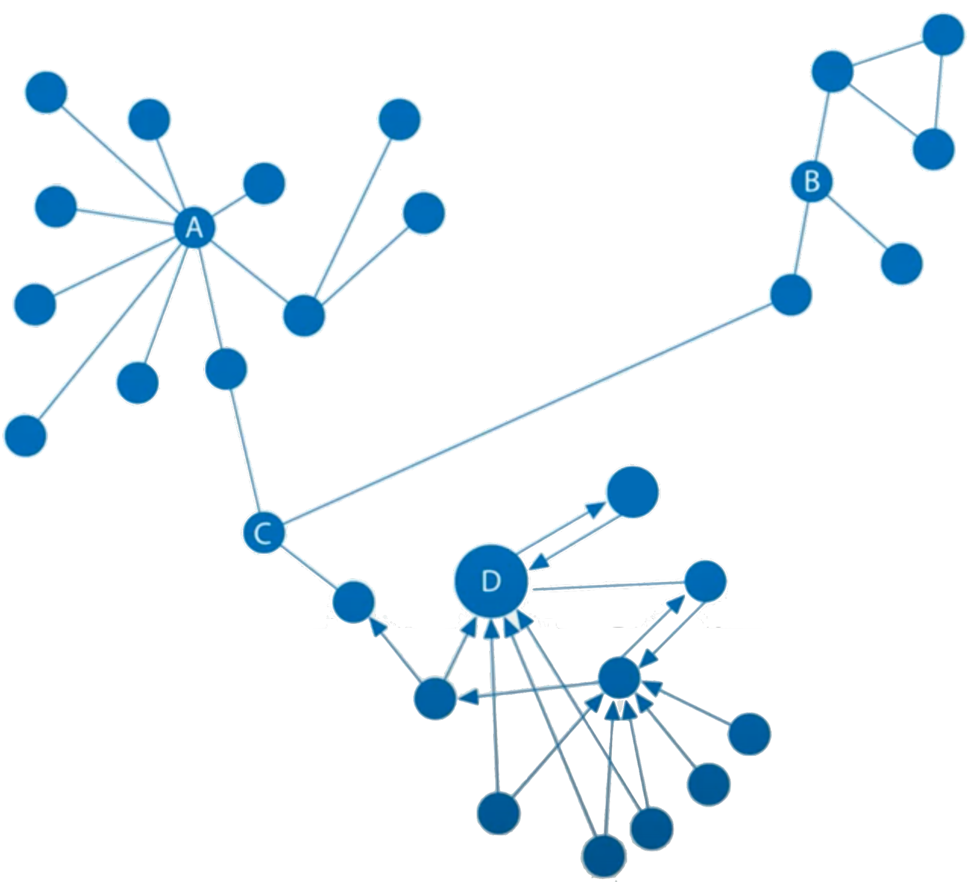

Figure 4.5: Sample Graph Exhibiting Different Topologies.

Figure 4.5 provides a sample graph with directed and undirected edges that exhibits some vertices identified by a letter. Each of these vertices has a high centrality measure according to a particular algorithm. The following list associates the centrality algorithms with these labelled vertices and present a formal definition of the computation of the centrality indices. While formal definitions of centrality formulas have many variations, the one presented in this list matches the implementation used in later section, in particular the formulas supported by Neo4j.

Vertex A: **Has the highest Degree Centrality ranking:** The degree centrality measures, for all vertices in the graph, their degree. This term is directly linked with the term *degree* or *valence* from graph theory which denotes the number of neighbors of a particular vertex. In directed networks, it is possible to restrict to the in-degree or the out-degree for detecting the most requested or the most communicative node in the network. The degree centrality assess the importance of a vertex based on the number of neighbours it has. A formal

definition of the calculation of the degree centrality for a vertex $u \in V$ may simply be expressed as:

$$\mathcal{C}_d(u) = k_u$$

Vertex B: **Has the highest Closeness Centrality ranking:** The closeness centrality measures the significance of a vertex through its distances to every other vertices in the graph. The closer the vertex is to the others, the higher its closeness centrality score will be. With the closeness centrality measure, central vertices are at a close distance to every other vertex in the graph. Unlike the degree centrality, the closeness centrality defines the importance of each vertex by the distance it has with every other vertex of the network. The closeness centrality can be computed for each vertex $u \in V$ following this formula:

$$\mathcal{C}_c(u) = \frac{|V| - 1}{\sum_{\substack{\forall v \in V \\ u \neq v}} d(u, v)}$$

This formula requires, for each node in the network to find the distance with every other nodes, that is computing the shortest paths. Therefore, computing the closeness centrality may be an expensive operation on large graphs, which led to approximation methods like [Saxena 2017].

Vertex C: **Has the highest Betweenness Centrality ranking:** The betweenness centrality defines the importance of a vertex based on the total number of shortest paths between any given couple of vertices passing through this vertex. Betweenness centrality is known to exhibit so-called bridges vertices which are nodes that help linking different communities. The computation of the betweenness centrality score for a given vertex $u \in V$ is expressed as the following:

$$\mathcal{C}_b(u) = \sum_{s \neq t \neq u} \frac{\sigma_{st}(u)}{\sigma_{st}}$$

Like for the closeness centrality, computing the betweenness centrality score requires expensive computations involving the identification of shortest paths between every couple of vertices in a given graph. There is also ongoing work for making fast approximation of the betweenness centrality score for large-scale graphs [Maurya 2019].

Vertex D: **Has the highest Centrality ranking with the PageRank algorithm:** The PageRank Algorithm is one of the variants of the eigenvector algorithm which was popularized by the search engine Google [Brin 1998]. In general, eigenvector algorithm assess the importance of a node based on the importance of its neighbors. An index is assigned to each vertices based on the number of connections it has to other highly connected vertices.

$$\mathcal{C}_e(u) = \frac{1}{\lambda} \sum_{t \in M(u)} \mathcal{C}_e(t)$$

where λ is a constant called the eigenvalue so that, with A_G being the adjacency matrix of graph G, $A_G X = \lambda X$. As of today, this algorithm is still

heavily used by the company and much work is achieved for optimizing its performance [Stergiou 2020].

4.3.2 Distributed Applications Bottlenecks

While bottleneck and chokepoints identification is still an hot topic in research, in particular when applied to the context of Cloud Applications, there is not a precise definition that emerged from literature of a bottleneck. In [Gan 2018b], authors propose to identify QoS violations that put an increased pressure on delivering predictable performance, as dependencies between microservices mean that a single misbehaving service can cause cascading QoS violations across the system. In actual literature [Ibidunmoye 2015, Marvasti 2013, Veeraraghavan 2016, Veeraraghavan 2018], bottlenecks and chokepoints are often detected based on service latency measurements and are refereed as QoS violations.

In this literature, bottlenecks often take one out of these two forms:

- The first one being the **resource saturation bottleneck**, it manifests when a single component reaches its *limits*. While the kind of limit may vary according to the type of service described, these limits may be CPU, memory usage, disk queue or rate limits of requests to an external API; it always causes significant delays to requests processing. Common methods to handle these bottlenecks are the use of message queues that can handle the back-pressure or the use of a dynamic scaling of such critical resources. Also, detecting these saturation bottlenecks is part of standalone monitoring and tracing data may not be game-changer in detecting this kind of bottlenecks.

- The second form of bottleneck is **the resource contention bottleneck**. It manifests in environments having semaphores, messages queues, buffers and mutexes. All of these mechanisms are software constructs commonly used in distributed computing, they are notoriously difficult to investigate on. Distributed tracing motivation has been providing data to ease investigation on these distributed mechanisms, to grant the administrator better insight on the application. This type of bottleneck is often explained by a poor software design that does not leverage the scalability and the workload distribution provided by the platform. Still, this kind of bottleneck is notoriously difficult to find and to investigate on and distributed tracing only provides raw data on components interactions.

The contention bottlenecks cause performance anomalies which usually result into saturation, deadlocks or partial failures of the system [Ibidunmoye 2015]. However, their causes can be numerous : misconfigurations or bad system tuning can greatly decrease performance, applications updates and introduction of buggy code can also lead to performances issues on the short term. Finally, underlying transient events or platforms re-configurations are common events that may cause bottlenecks within the system.

As the contention bottleneck are difficult to detect at runtime, we propose using the global application networking model to spot the chokepoints in a Cloud application. We materialize the chokepoints as a potential bottleneck whose failure could

greatly impact the behaviour of the application. The **Betweenness centrality algorithm** is a promising candidate to identify chokepoint application as it main interest is to identify the vertices in the graphs involved in the highest number of path between any couple of vertices. Vertices with a high betweenness centrality score are involved in multiple services compositions and are more likely to cause congestion when the load will increase.

The following section details the instrumentation of the simulation program named *Spigo* that has been used to create the microservices graph presented in figure 1.2 of chapter 1. Its capability of creating large-scale application simulations will be used to create complex communication graphs. Then, by computing the centrality of each of the vertices of the graph, we will be able to quickly identify chokepoints and rank services.

4.4 Implementation

In order to generate a graph representing a massively distributed application, a simulation program has been used. Indeed the cost of a Zonal Kubernetes Cluster in GCE was already high and the representativeness of this platform for bigger application was not sufficient. Therefore, some experiments have been made with the software *Spigo*[4] that has been used by A. Cockcroft to present its workshop on large scale cloud infrastructures [Cockcroft 2016a, Cockcroft 2016b].

Spigo is designed after the *Actor Oriented Design Pattern* [Lee 2003] that revolves around the concept of message passing and communication among the *so-called actors* of the system. Actor can be compared to threads as they are independent processing units within the program that react to incoming messages. Actors life-cycle is also managed by a high level scheduler and actors are also organized following a hierarchy. Instead of communicating over the network, actor communicate via pipes which make communications faster. By its construct, the actor design pattern is extremely close to the microservices architecture. The *Spigo environment* is coded in the language Go: it leverages the lightweight implementation of threads named *go-routines* to run hundreds of actors simultaneously. Each of these *go-routines* model a microservice that communicate with the over services of the application. The communication is done through *channels* another distinguishing point of the go language that defines communication pipes between the asynchronous elements of the application.

Spigo is bundled with existing scenarios defining several classical architectures often used when building a web application on AWS. Some examples presented in the environment are: the 3-tiers Architecture with Apache - PHP - Database, its evolution with AWS components, a Big Data Pipeline, some Netflix-Inspired architectures, a Distributed Timeseries Database for IoT inspired by the service *Riak*[5] and also a Flying Spaghetti Monster. While not being actively maintained

[4] `https://github.com/adrianco/spigo` Public repositoriy of the code the spigo simulation program hosted on GitHub.

[5] `https://riak.com/products/riak-ts/` a NoSQL database physically distributed ingesting and enriching IoT Data

anymore by its creator, *Spigo* still constitutes an opportunity to emulate a multi data centre Cloud Application in a standalone computer. In this section, a focus is set on the usage of this environment to emulate a fully distributed application and emit traces compliant with the OpenTelemetry format. Then, the simulation of a large scale application is presented along with the choice of the scenario. Finally, the centrality of the graph resulting is analysed and we illustrate the use of the betweenness centrality to spot chokepoint in an application.

4.4.1 Using Spigo for Emitting OpenTelemetry Traces

By default, Spigo does not emit traces in the OpenTelemetry format, still it is capable of creating a graph representing components interactions. This graph does not match the model defined in chapter 2 and only represents microservices in a single communication layer. The initial work aimed to instrument *Spigo* code base with the OpenTelemetry instrumentation library: while it provided a real OpenTelemetry compliant data for the simulation, the Go environment was old and did not allow to freeze version of the instrumentation library of OpenTelemetry. This lead to heavy work on the code base and unpredictable behaviours at compile time of the simulation program. Instead of working on the migration of *Spigo* code base to a more sustainable eco-system, we used *Spigo* connection with Neo4j and applied multiple *Cypher* requests to rebuild the hierarchical model used throughout this thesis.

Spigo graph model behaves exactly like the initial graph models representing traces: the model is not hierarchical and vertices hold a dense amount of metadata representing the resources. Listings 6 and 7 present the *Cypher* queries to recreate the hierarchical model from the metadata held in each vertices in the graph. Indeed, in listing 6, at line 2 of the query, metadata are extracted into the variables `nodeId`, `regionName` and `zoneName` to be later used to create new vertices, if they do not already exist. Also the last 5 lines of the requests add the relations to the newly created vertices. Finally, listing 7 is the implementation of the graph rewriting process presented in section 2.4.2.1 but in the *Cypher* language.

```
1  MATCH (n:riak)
2  WITH n, n.ip AS nodeId, n.region AS regionName, n.zone AS zoneName
3  MERGE (vm:VirtualMachine {ip: nodeId})
4  MERGE (zone:Zone {name: regionName + '-' + zoneName})
5  MERGE (region:Region {name: regionName})
6  MERGE (n)-[:EXECUTES_ON]->(vm)
7  MERGE (n)-[:EXECUTES_ON]->(zone)
8  MERGE (n)-[:EXECUTES_ON]->(region)
9  MERGE (vm)-[:IS_CONTAINED]->(zone)
10 MERGE (zone)-[:IS_CONTAINED]->(region)
```

Listing 6: Creation of the Hierarchical Layers

After executing these requests on the graph build by the *Spigo environment* the

```
1 MATCH (r_src)<-[:EXECUTES_ON]-(src:riak)-[:CONN]->
2       (dst:riak)-[:EXECUTES_ON]->(r_dst)
3 WHERE labels(r_src) = labels(r_dst)
4   AND r_src <> r_dst
5 MERGE (r_src)-[r:PROJECTED_REF]->(r_dst)
6   ON CREATE SET r.total = 1
7   ON MATCH  SET r.total = r.total+1
```

Listing 7: Projection of the Hierarchical Layers

graph stored in Neo4j follows the hierarchical model described earlier. Still, in these two requests the vertices created by the simulation are labeled after the name of the simulation, these requests have been run on the Riak IoT scenario. In the following section this scenario is presented along with its chokepoints.

4.4.2 Scenario Selection and Representativeness

Spigo has several scenario bundled within the application; to represent a real-world Cloud-IoT application, the scenario modelling the Riak distributed database was identified to be the most suitable to the research area presented in this thesis. Indeed, this scenario is based on Riak application which is a distributed and resilient database targeted for IoT needs. Its structure is heavily hierarchical and leverages the geographical distribution of computing resources to be as close as possible to IoT devices. Figure 4.6 represents a logical chaining of the components of the application. Table 4.2 details the signification of the symbols used in the figure.

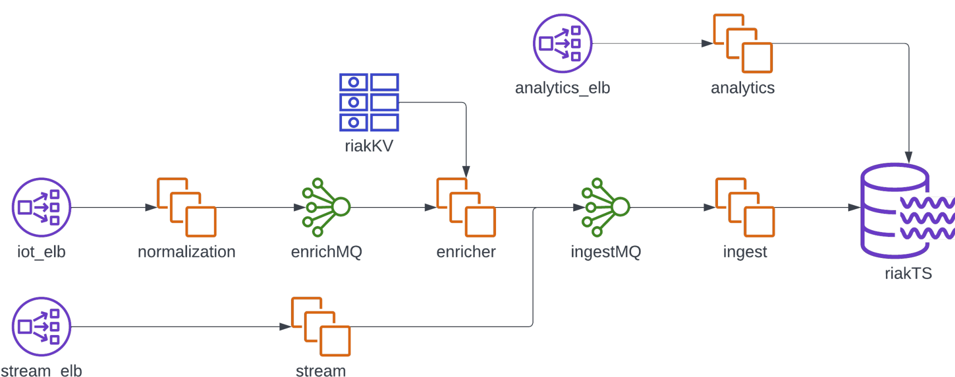

Figure 4.6: Logical architecture of AWS components in the Riak Scenario

The use case of the Riak application provides an implementation of an heavily hierarchical app, having multiple endpoints scattered across zones and positioned close to the users. All these endpoints, materialized by the load-balancers converge to processing data to write in the same common database. This Riak scenario

Symbol	Meaning
	This symbol represent a traffic load-balancer. The spigo simulation environment designates these components with ELB which stands for *Elastic Load Balancer*. Usually, in AWS architecture, an ELB is an entity that routes external traffic to the services in the application.
	This symbol represents generic instances of a service capable of scaling. This is a generic symbol that can model almost any replicated services whether being in the same zone or in different regions.
	This symbol represents a channel in AWS architecture diagrams. In this simulation environment, in particular, they represent message queues, a building block used to create data processing pipelines.
	This symbol represents Key-value store, a trivial database designed for intensives reading operations. In this simulation scenario, it represents the dictionary used for enriching IoT messages with custom context. In a geographically distributed context Key-Value Stores are scattered in multiple zones and the service reading values request the entity in the same zone.
	This symbol represents Data-Lakes. For this simulation they are materialized by a Cassandra Cluster, a distributed Zone-aware database that ensure the availability of data through replication. While the Cassandra Database is aware of the location of the instance, this setting is not used for optimizing traffic but only for ensuring availability of data.

Table 4.2: Legend of AWS symbols

follows a conventional pattern of data-processing pipline architecture. Such data pipelines are extremely vulnerable to bottlenecks; and in particular, to resource contention bottlenecks. A single stage failing in these pipelines can take down the whole system and result to data loss or SLA violations. This is the role of the message queues in this architecture, to ease the load-balancing of the resources creating data and consuming data. The service *ingestMQ*, in particular, plays a critical role and aggregates data from two distinct endpoints and prepare it to be consumed by the *ingest* microservice.

This architecture in *Spigo* is configured in a JSON file named after the scenario and listing for each microservice. The scenario's name, template, the number of replicas the application has within a region and, finally, its dependencies are all parametters that can be tuned. To replicate this architecture into multiple regions when running the simulation environment, an option has to be specified at launch. In a single region, both *ingestMQ* and *enrichMQ* are replicated three times, still *ingestMQ* has to deal with more requests and have to undergo more pressure because of the merge of the processing pipelines of `iot_elb` and `stream_elb`.

Figure 4.7 presents an excerpt of the communication graph of services of a simulated riak application whose components are replicated multiple times and scattered in three zones. This results in 135 microservices communicating over three zones. In this figure, each color of the vertices in the graph represents one of the services presented in figure 4.6. A focus is set in the *ingestMQ* instances and their bridge role in the application.

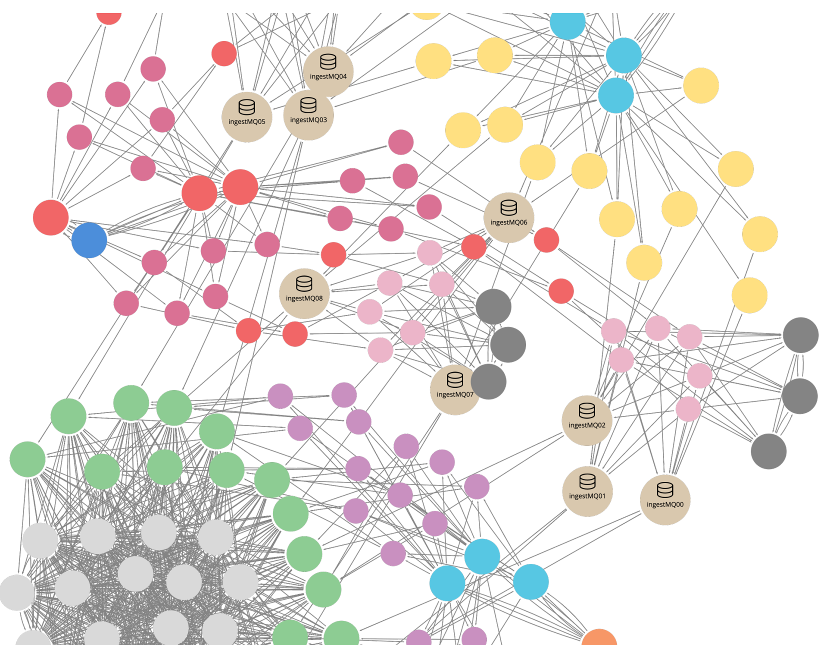

Figure 4.7: A sample of the graph of services with a focus in *ingestMQ* services

4.4.3 Observing the Impact of Betweenness Centrality in the Riak Simulation

The simulated application of 135 microservices scattered over three zones creates a big graph that cannot be simply processed by human mind, unlike the global riak architecture overview. The service *ingestMQ* was identified as a potential chokepoint of the application through an architectural analysis. In order to make instances of this service less critical, a trivial solution is to increase the number of replicas: the load of requests handled by each instance of the service would become less important and, at the scale of the application the chokepoint would be less important. Therefore we will focus making the service *ingestMQ* scale up and down to observe the effect on global graph centrality scores. In the global architecture, *ingestMQ* is a three-replicas message queue deployed in each zone to serve local users. By its role and its design, this service is concerned by risk of resource contention bottleneck.

To compute centrality indices of each vertices, we relied on the *graph analytic library*[6] of the Neo4j database. Figure 4.8 displays the graph of services and the size

[6] `https://neo4j.com/product/graph-data-science/` a full set of graph algorithms that can be

of vertices represents their betweenness centrality score. Vertices highlighted in blue represents the *ingestMQ instances*. Table 4.3 aggregates the results of the centrality computation and provides a per-service range of the centrality scores obtained by the betweenness centrality algorithm.

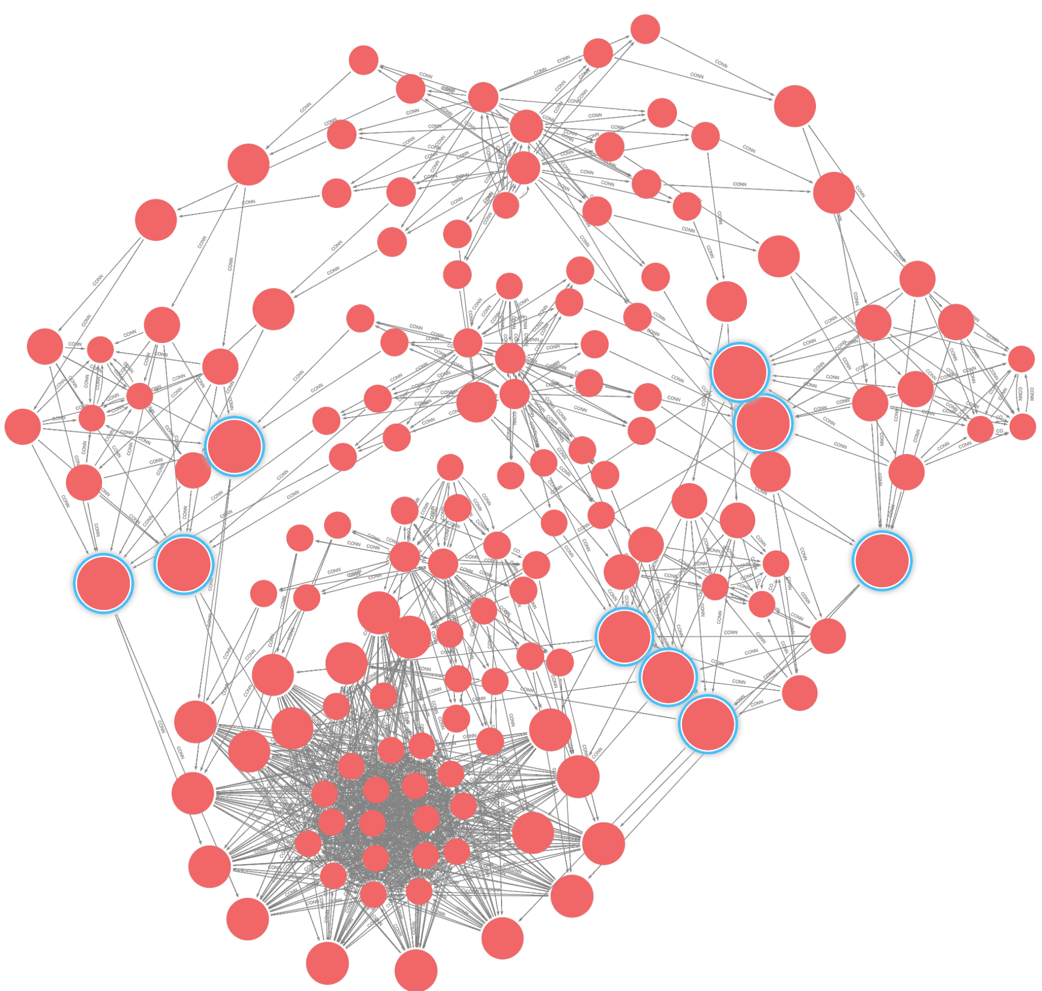

Figure 4.8: Cloud Application Graph Visualization Where Vertices Size is Proportional to the Betweenness Centrality Score of the Vertex

run on a Neo4j graph

Service Name	Replicas	$\mathcal{C}_b$
`ingestMQ`	9	$[181, 193]$
`ingester`	18	$[97, 104]$
`enrichMQ`	9	$[87, 97]$
`enricher`	18	$[53, 57]$
`normalization`	18	$[13, 18]$
`stream`	18	$[3, 12]$
`analytics`	18	$[5, 9]$
`riakTS`	18	$[0, 0]$
`riakKV`	9	$[0, 0]$

Table 4.3: Betweenness Centrality score range for each group of services

The betweenness centrality calculation clearly identifies the service *ingestMQ* as the most critical service along with the *ingester* service. Each *ingestMQ* has a betweenness centrality score between 181 and 193 according to this formula (not normalized), this score outshines other betweenness centrality score of any other vertices in the graph. The betweenness centrality score grants a risk indicator of the impact of a failure from these service. As the number of instance increases the impact of a failure of these component should decrease.

To verify this assertion, the simulation has been executed by changing the number of replicas of the *ingestMQ* service. In a first scenario, the number of instances of the *ingestMQ* services has been lowered to one per zone, and, in another scenario it has been scaled up to 5 instances per regions. Both figure 4.9 and table 4.4 shows the results of the simulation. The figure shows the two resulting graphs where the size of the vertices depends on its centrality score, also *ingestMQ* instances are highlighted in each graph. In the table 4.4, column $\mathcal{C}_{b1}$ gives centrality score for the whole graph when there is only one instance of *ingestMQ* service per zone. Column $\mathcal{C}_{b3}$ keeps the previous values and $\mathcal{C}_{b5}$ provides the score when there are five instances of the service *ingestMQ* per zones.

Service Name	$\mathcal{C}_{b1}$	$\mathcal{C}_{b3}$	$\mathcal{C}_{b5}$
`ingestMQ`	$[383, 422]$	$[181, 193]$	$[104, 151]$
`ingester`	$[187, 219]$	$[97, 104]$	$[118, 123]$
`enrichMQ`	$[74, 77]$	$[87, 97]$	$[92, 110]$
`enricher`	$[43, 44]$	$[53, 57]$	$[57, 65]$
`normalization`	$[14, 16]$	$[13, 18]$	$[13, 25]$
`stream`	$[3, 7]$	$[3, 12]$	$[8, 12]$
`analytics`	$[5, 10]$	$[5, 9]$	$[5, 9]$
`riakTS`	$[0, 0]$	$[0, 0]$	$[0, 0]$
`riakKV`	$[0, 0]$	$[0, 0]$	$[0, 0]$

Table 4.4: Betweenness centrality score range for each group of services with a varying number of *ingestMQ* instances

As a result, we can observe that the betweenness centrality score drastically increases when the service ingestMQ is only present once per region whereas it

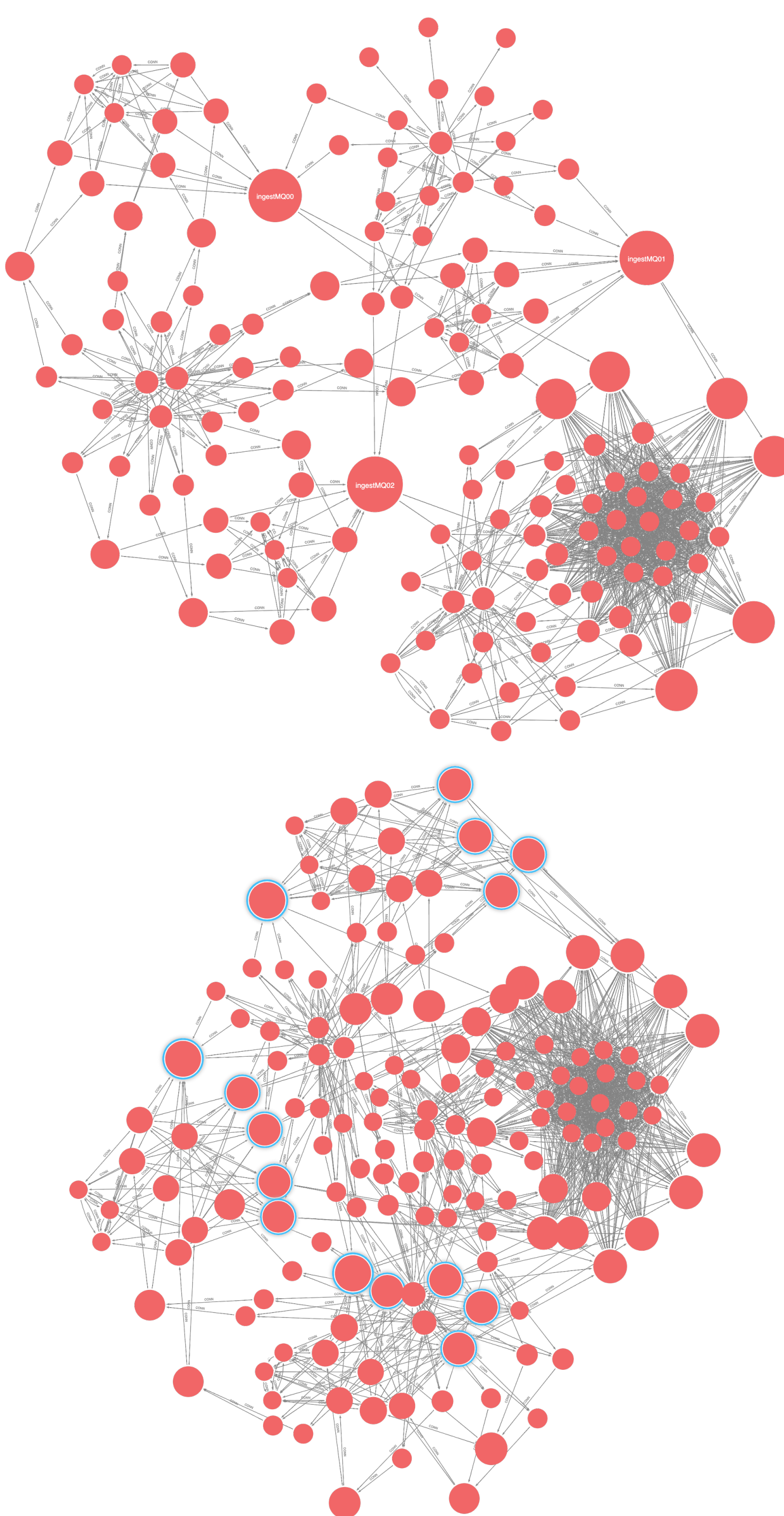

Figure 4.9: Cloud Application Graph Visualization respectively with one and five instances of *ingestMQ* per zones

confounds with other scores when there are fives instances of this service per zones. On a communication graph, computing the betweenness centrality on nodes has the capability of exhibiting chokepoints, and while the chokepoints exhibited in this example was solved by scaling up a particular service, in real world scenario solution may not seems to be as trivial as this. Indeed, message queues like *Kafka*[7], *RabbitMQ*[8] or *NATS*[9] are stateful services and scaling them up and down is a costly operation as it may require to re-think the sharding and the duplication of data among instances.

Also, in this experimentation, we stayed at the first layer of the hierarchical communication graph and none of the higher layers has been considered in this experimentation. The reason behind this choice is because in the Netflix architecture, and unlike in Kubernetes, the placement decision in zones and regions is made by a human architect. The simulation environment does not have the representativeness to do a numerical study and projecting the communications relationships observed between services among zones and region quickly create a complete graph that is not representative of a real-world behaviour of a Cloud application.

4.5 Conclusions

In this chapter, a focus has been set on using the communication graph maintained by the trace processing pipeline instead of using each trace individually. The communication graph maintained by the trace encoding pipeline has the capability of exhibiting chokepoint in a large scale architecture, which address a problem commonly encountered in massively distributed system monitoring. Centrality algorithms have been proved to have the capability of addressing these challenges. We addressed the generality of both the approach and the encoding pipeline by modelling a Cloud Application made on AWS and following the Netflix architecture.

Throughout this chapter, we leveraged the *Spigo* simulation program capable of emulating a physically distributed application that follows the architecture principles of Netflix. These architecture principles result in consequent graphs of intertwined services that are difficult to process for a human operator. With this contribution, we propose the use of the betweenness centrality to spot chokepoints in a massive service graph and therefore identify the critical assets of the system, that cannot be identified with standalone system monitoring.

The work in this chapter reviewing the use of the simulation environment and the centrality analysis has been published in [Cassé 2022].

This simulation program used throughout this chapter only represents components interactions and does aim to be also representative of network latency of the system or of dynamic placements of resources. Also, the *Spigo* software is not maintained anymore by its creator and its motivation was to represent interactions on the building blocks of the Netflix architectural principles, which does not match State-of-the-Art design principles for nowadays Cloud applications. For this rea-

[7]https://kafka.apache.org
[8]https://www.rabbitmq.com
[9]https://nats.io

son, the significance of this experiments can only stay on the qualitative aspect of the monitoring but not on the quantitative aspect. Data from a real application can exhibit important numerical measures: these measurements like the number of bytes exchanged or the network latency, would allow to annotate edges in the model with numerical scores.

Conclusion & Future Works

Contents

Synthesis of Contributions **104**
Future Works **105**
Short-term Work 106
Mid-term Work 108
Long-term Work 109
Closing Words **109**

Over the last two decades, the adoption of Cloud Computing has had significant impacts on the way we now think of software performance analysis, in particular when hosted in a Cloud platform. Whereas the Cloud Computing paradigm eased large scale application development, deployment and maintenance, it also hid software execution, gave less control to developers to instrument their code and considerably obscured debugging and performance analysis. Also, Cloud applications embraced the distributed system architecture, which solved multiple scaling issues. It also raised a new spectrum of challenges for software monitoring like spotting partial failures, bottleneck or raising consistency issues. These two drastic software shape changes lead to a wide corpus of literature tackling these monitoring challenges under multiple different angles. Multiple contributions addressed the scaling problem of metric gathering, or the heterogeneity of monitoring data in a cloud application. Also, root-cause analysis became increasingly important as the system grew, as well as detecting and observing dependencies in a large and volatile system.

In this thesis, we have provided a study on Cloud Application performance from the perspective of distributed system monitoring. This study is supported by monitoring data gathered by *OpenTelemetry*, the last innovation brought by the open source community, being increasingly adopted within the industry. With this new monitoring standard, the monitoring of Cloud application has undergone massive changes: by leveraging context propagation in a distributed application, the recent monitoring tools are capable of aggregating latency measurements from multiple services to recreate a “distributed trace”. These traces carry a unique information that describes component interactions in the whole application. The discovery of these services interaction is based on the observation of the system, and aims to have a minimal performance footprint, suitable for an industrial use. Despite this new kind of monitoring data becoming more and more popular among Cloud developers, no consensus among the tech industry and the academic has been observed on the usage of these traces.

A review of initiatives from the large tech industries highlighted the potential of traces for investigating performance anomalies in massively distributed applications, in particular addressing these new distributed system challenges. Still these usages have been kept under closed sources by the industry actors, whether being the trace

format or their usage. The *OpenTelemetry* format, that emerged during the thesis, opened the door of a more generic approach for observing and analysing distributed applications under a different angle. The driving idea in this thesis has been to focus on purely distributed performances challenges where the observation of components interactions could bring valuable insights on the overall application performance.

The State-of-the-Art of distributed application challenges raised that Cloud application hosting is not fully centralized anymore. Indeed, applications are scattered among multiple data centre around the world, and, therefore the network linking the components of the application cannot be considered even. By leveraging *OpenTelemetry* semantic, we have identified that distributed tracing offers the potential of observing and highlighting the different types of communication in Cloud application. The contributions of this thesis centered around an *OpenTelemetry* trace encoding model that exhibits the overall application structure. We proposed a new usage of distributed tracing in geographically distributed application having an uneven scattering of servers to spot inefficient service composition and bottlenecks.

While these scenarios and use-cases may appear to be quite niche today, we witnessed, over the past years, an increased adoption of Edge computing, Cloudlets, IoT or Points of Presence (for streaming) to increase application performance. All of these concepts refer to the expansion of the Cloud toward the end-users, and the model proposed in this thesis can easily fit these use-cases as long as the metadata is correctly set in traces.

Synthesis of Contributions

Throughout this thesis we provided a study on Cloud Application performance through the angle of distributed system monitoring. This study addressed the following points:

- A literature review focused on Cloud environment, addressing both architectural challenges from a provider perspective and observability challenges for the customer perspective. This study pointed out the need of a more evolutionary Cloud Computing model and the impact of the adoption of the distributed structure for monitoring, in particular at large scale. Also, in this study a particular attention has been brought to the different initiative for untangling dependencies and building a high level view of the application.

- The definition of an encoding model for *OpenTelemetry* traces that rebuilds a global vision of a distributed application by focusing on its architecture and component interactions. This model has been proved to be adaptive to fit multiple Cloud-based architectures and relies on a well-defined semantic and tooling.

- A proposition to use this model in a physically distributed application to identify the cycles that can induce latency and an extra cost when composing services from multiple zones in Zonal Kubernetes Clusters. This study has been based on the need expressed by the team in Orange in charge of developing *Djingo* (a voice assistant) to have better observability on inefficient service

composition and their correlation with the overall application performance. Unfortunately, the development of the voice assistant has witnessed multiple changes, and the problem initially formulated by the architectural team was not applicable in the last iteration of the product.

- An other proposition to use this model to spot chokepoints with centrality algorithm in a large graph of services. Instead of processing each trace individually, this proposition considers the accumulation of traces to recreate a large graph of microservices interactions. A simulation environment was used to recreate a large scale application, this program uses threads and messages queues to mimic *micro services* and *network calls*. This simulation demonstrates the use of the betweenness centrality algorithm to automatically detect the chokepoints in a graph of micro services. Whereas this scenario was not motivated by the *Djingo* environment, the detection of chokepoint within a large and complex architecture was among the motivations that lead Orange to this thesis.

In this thesis, an effort was made to address the challenges unique to Cloud Application monitoring by proposing a model and an implementation based on real-world tooling and constraints. Indeed, the trace encoding pipeline presented in chapter 2 suppose the use of *Jaeger*, the de-facto tool for visualizing distributed traces. The encoding pipeline is coded in the Scala language and has the capacity of being executed in parallel. The implementations have all been realized with massive data processing constraints in mind and are capable of being executed on a production-ready *Jaeger* instance.

The recent evolution of *OpenTelemetry* has provided an environment to obtain real-world measurements from an application and therefore enabled the capability to have a quantitative study. *OpenTelemetry* specifications of traces and metrics went stable during the redaction of this thesis, the specification of log messages and their semantic are still in Beta at the time of writing. Two immediate extensions of the work presented in this thesis would be to process a flow of traces of an application in production:

- Plug the pipeline on an application instrumented with *OpenTelemetry* and evaluate the results with a quantitative study.

- Evaluate sparse sampling of traces to not overwhelm the system with a massive amount of data, like the work presented in [Las-Casas 2018].

In the next section, we discuss on the possible extensions of the work presented in this thesis.

Future Works

The two different usages of the model proposed in chapter 3 and 4 are not based on a real flow of traces from a fully fleshed cloud application. Both of these use-cases require a large scale application instrumented to emit traces; also, *OpenTelemetry* is

a recent initiative which started in 2019 and that stayed in beta for a long time. The production constraints on *Djingo* and the development timeline of *OpenTelemetry* were not compatible and the traces were only used in a development environments. For this reasons, the experiment presented in this thesis have been realized on small scale clusters not receiving a real flow of user requests.

As Cloud performances anomalies are still difficult to represent, the implementations presented in this thesis have been focused on a qualitative approach instead of a quantitative approach. Since then, *OpenTelemetry* gained in maturity and now has stable releases, newly developed applications can use this format to emit and process traces. Also instrumentation libraries are expected to add multiple fields in traces that would enable a numerical study, like for example, the number of bytes in a network communication between two microservices. During the PhD, we witnessed an strong adoption of this monitoring format within the cloud communities, the amount of instrumentation libraries presuppose of a rich eco-system and a strong use by the industry, in particular for newly created software.

Still, a quantitative study will have to wait for a real data-source, from an application geographically distributed and instrumented with *OpenTelemetry*. In this section we present potential extensions of the works presented in this thesis: in the short-term work we present a data visualization that leverages the encoding model and the hierarchies to provide a global view of a distributed application. An expressive visualization could improve adoption of the distributed tracing technology in production environment. Also in Mid-term work and later we will presuppose of an adoption of the technology to achieve quantitative studies.

Short-term Work

Traces have the potential of exhibiting components interactions in a Cloud Application, still there are almost no visualization tools that allow a human operators to grasp the big picture of an application. Whether we consider *Jaeger* or dashboards provided by Observability Platforms like *DataDog*, *Lightbend* or *Grafana*, there are no application wide visualization representing the application as a whole. However, the model presented in chapter 2 aims to exhibit complex placement and distribution of computing resources, and the application presented in chapter 3 focus on inefficient communication at higher abstraction levels. In [Holten 2006], authors define the “Edge Bundling Visualization” that represents the adjacency relations among entities organised in hierarchy. This visualizations focuses on the communications represented as adjacency relations, and on the number of hierarchy layers crossed for the communication of these entities. The hierarchy is represented as a radial tree, also called a *flare*, where the root entity is situated at the center of the flare. The further away we go from the center, we have the intermediate layers of the hierarchy until the leaf elements of the hierarchy situated at the end of the circle.

Figure 5.1 provides an example of the representation of a communication between two entities part of a hierarchy. On the left, there is a normal representation of the adjacency relation with a strait line not displaying the number of layers crossed by this communication. On the right, the line has been curved to follow the

branch of the hierarchical tree which represents that the entities communicating do not share common layers. These adjacency relations are represented with a tension toward the center that is more important for entities not sharing common parent elements in the hierarchy.

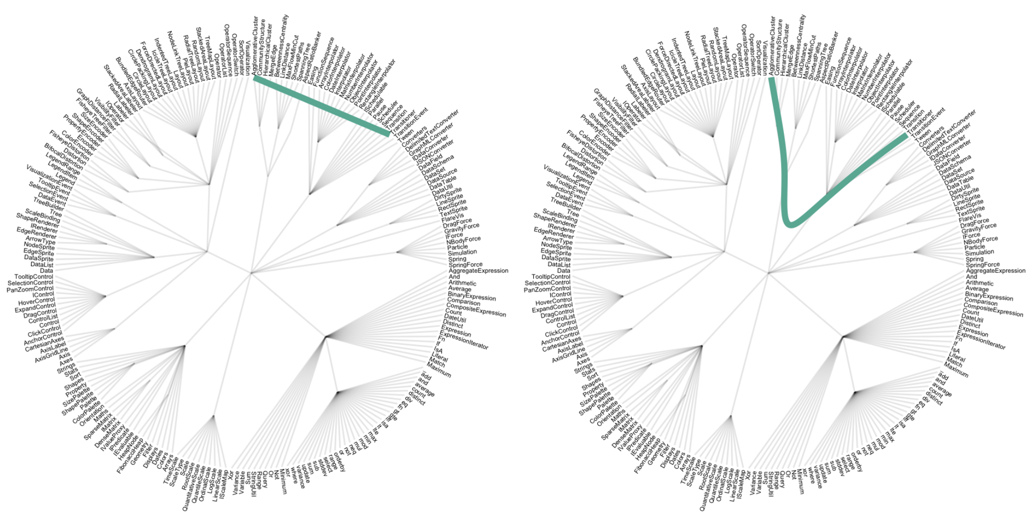

Figure 5.1: Behaviour Principle of the Hierarchical Edge Bundling Visualization: the Tension of the Arcs Depend on the Number of Layers Traversed

This visualization is particularly suitable for representing complex Cloud Application like Zonal Kubernetes Cluster. In that case, the hierarchy in the flare would be the physical positioning of resources and the representation of adjacency relations would show the amount of communications that cross the Zonal boundaries and that are added to the cloud bill. Some software like *infra-scrapper*[10] have already used Edge Bundling Visualization to represent component interactions like shown in figure 5.2, still the hierarchy used in the flare was not a representing the physical location but the logical embedding of entities within Kubernetes. Therefore the representation only displayed the interactions between Kubernetes namespaces, which does not bring much value to the representation. Instead, with the hierarchy inside the flare representing the geolocation of resources, the visualization would highlight the amount of communication crossing the boundaries of servers, data centres, availability zones or even regions.

Bringing this kind of visualization to *Jaeger* help to use tracing not only as a debugging tool but also as a dashboarding tool to provide a higher view of a Cloud Application. Implementing this visualization tool would provide another bridge between the model from chapter 2 and the implementation proposed in chapter 3

[10]`https://github.com/cznewt/infra-scraper/` Open source tool for analysing the infrastructure of a Cloud Application based on Kubernetes

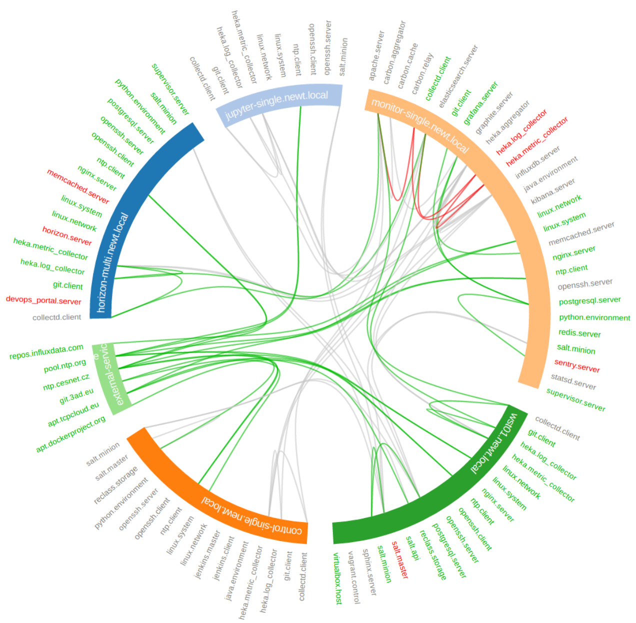

Figure 5.2: Example of Edge Bundling Visualization provided by *infra-scrapper* to represent Pod Interaction in a Kubernetes Namespace

by providing another way of displaying the costly communication an application. This visualization would also benefit from the gain in maturity of tooling and the addition of fields, as it would allow to set a thickness to the adjacent relation like, for example the number of bytes exchanged.

Mid-Term Work

Once the adoption of distributed tracing within the industry will settle down, gathering traces representative of real world performance issues will be possible. This would enable quantitative analysis on tracing data, and therefore to add weight to the flow hierarchy metric calculation. In addition, adding weights to the model would apply to the global application model where the betweenness centrality was used to detect chokepoints. Both ideas leveraging the model have been presented with a weighted and unweighted approach. The unweighted approach has been preferred for a qualitative approach, still with time, a quantitative approach would provide more accurate results.

Chapter 3 presented the use of the flow hierarchy metric that counts the number of edges not involved in a cycle, and the impact of cycles in a Cloud Application. Also, in [Zafeiris 2018], authors mention multiple other metrics related to flow hierarchies that could find a use with the proposed model. In particular, authors propose a metric that would complete the flow hierarchy: it consists on the minimum fraction of edges to be removed to make the graph cycle-free. This approach would complete the flow hierarchy to quantify the number of inefficient communications occurring when serving a request.

Also, Chapter 4 presented the use of the betweenness centrality to detect choke-

points in the application. The detection is note performed online and require to recompute the entire graph at each computations. Also computing centrality scores over the shortest path between any given vertices may not be representative of all the kind of communications occurring between microservices. It would be possible to enable an online computation of a scoring inspired y the betweenness centrality where, instead of iterating over every shortest paths in the graph; iterations would be performed over each traces collected from Jaeger.

Long-Term Work

Over this thesis we used distributed tracing to take another approach to distributed application monitoring. A focus has been made on scenarios an use-cases where observing the dependencies at runtime would bring a high value. In a later future, we could imagine the extension of the technology used by traces to observe communications not only for application performance monitoring but also to a wider variety of usages. For example, IT security relies on logging and networking probes to rebuild a graph of entities taking part in the information system of the company. Adding context propagation in the networking devices and policies of the compagny could enable to recreate traces.

In IT security, these traces would also highlight communication behaviours between entities taking part in the system. Also companies structures are also inherently hierarchical: a multitude of hierarchies can be used to model interactions between devices, employees, or even access rights. In a different context, the graph rewriting approach that projects dependencies over higher abstraction layers can show an attacker trying to access an different kind of resources.

Closing Words

Throughout this thesis we have set the focus on Cloud application performance monitoring under the angle of distributed system anomalies by relying on a promising technology, still under heavy development. This technology takes the form of traces, displaying, in a Gantt chart, a vast amount of properties following a well-defined semantic. This data has a different usage than logs and metrics and has the capability of monitoring and observing dependencies in a distributed system at runtime. Actually mainly used for debugging purposes, in this work we propose different usages targeting production environments and scenarios.

This work has long been a part of the Djingo Project in Orange, the team has provided both the initial research challenges and data. But these last years have lead the team and Orange to review their priorities and unfortunately, the proposed model has not been able t reach production and to ingest a real flow of traces. Still, the challengesllenges addressed remain generic, and over the past years multiple other projects have tried and adopted OpenTelemetry as a framework for monitoring their application.

Appendix A

Résumé en Français

Contents

A.1 Monitoring d'Applications Cloud 114
A.1.1 Introduction 114
A.1.2 Présentation du Paradigme Cloud 114
A.1.3 Les Applications Cloud Natives 115
A.1.4 Monitoring et Analyse de Performance d'Application cloud . 116
A.2 Modélisation des Communications Internes 118
A.2.1 Introduction 118
A.2.2 Présentation de l'Écosystème de Tracing 118
A.2.3 Extraction et Aggregation des Données dans un Graphe de Propriétés 119
A.3 Détection de Composition de Services Inefficaces 123
A.3.1 Introduction 123
A.3.2 Modélisation d'une Application cloud grâce au Concept de Hiérarchies 123
A.3.3 Détection de Communications Inefficaces 124
A.3.4 Mise en œuvre 126
A.4 Détection de Goulots d'Étranglements 128
A.4.1 Introduction 128
A.4.2 Généralisation de l'Encodage en Graphe de Propriétés 128
A.4.3 Utilisation de l'analyse de Centralité pour l'Anticipation de Goulots d'Étranglement 130
A.4.4 Vérification Expérimentale 132
A.5 Conclusion 137
A.5.1 Synthèse des Contributions 137
A.5.2 Pistes de Poursuite des Travaux 138

Introduction

Contexte

Au cours des dernières années nous avons pu observer un essor des Applications Web dans notre quotidien. Qu'il s'agisse de réseaux sociaux, de sites marchands ou de plateforme de diffusion de contenu multimédia, tous ces cas d'utilisation prennent la forme d'applications Web. Et bien que la croissance de ces applications était déjà importante, la crise du COVID-19 a accéléré cette expansion. Des études telles que [Cantor 2020] ont démontré, lors du confinement, que YouTube drainait, à lui seul, plus de 15% du trafic mondial juste devant Netflix qui lui en occupait 11%. Cet exemple illustre l'avantage principal de ces applications : leur extrême *scalabilité*[1] : ces application sont capables de répondre à un nombre de requêtes utilisateur très changeant tout en maintenant un niveau de service élevé pour l'utilisateur.

Cette scalabilité est permise par le paradigme du Cloud Computing, apparu il y a une vingtaine d'année, qui permettait d'accéder quasiment instantanément à des ressources de calculs hébergées dans des centres de données. Depuis, ce paradigme a évolué et a radicalement changé les méthodes de développement de logiciels. Afin de tirer le meilleur parti des avantages du Cloud Computing, les développeur se sont adaptés et ont changé l'approche du développement d'applications. Les applications dites *Cloud Natives* sont conçues comme des systèmes distribués. Elles exploitent les capacités des systèmes distribués afin de maintenir un haut niveau de disponibilité, et de performanceet ce quel que soit la charge appliquée par les utilisateurs.

L'adoption du paradigme du Cloud Computing a aussi eu un impact sur les méthodologies de monitoring de ces applications. En effet, en tant que système distribué, les cause de pannes et de dégradations de services ont été démultipliées. Les systèmes de suivi de performance de ces applications doivent pouvoir détecter de nouveaux types de pannes telles que les goulots d'étranglement, les erreurs d'intégration ou même la vampirisation de ressources.

Environnement Industriel et Problématique

Les travaux présentés dans cette thèse font partie d'un projet de recherche initié par Orange Labs Services, la division de R&D du Groupe Orange et ont été réalisés avec la collaboration des équipes de conception de Djingo. Le projet Djingo consistait à concevoir un assistant vocal intégré à l'écosystème Orange. Les serveurs de Djingo avaient été conçu avec une approche Cloud Native et réparti sur plusieurs centres de donnés en Europe. En tant qu'assistant vocal, les performances de Djingo étaient critiques pour les équipes de développement. Cependant ce projet n'a pas été mené à terme et Orange, pour des raisons stratégiques, n'a pas commercialisé le produit et a arrété son développement en 2020. Malgré l'issue du projet, le cas de Djingo a permis d'illustrer des problématiques de performances liées aux application Cloud et a motivé les contributions présentées dans ce document.

[1] anglicisme du mot *"scalability"* communément utilisé dans le domaine de l'informatique pour caractériser la capacités d'un système à s'adapter a une changment important dans la charge qu'il gère.

J'ai donc mené des travaux pour formaliser une approche pour la détection d'anomalies de performances mais sous l'angle de l'analyse de traces. Ainsi, cette thèse propose une approche de la détection problèmes de performances au sein d'une application Cloud sous l'angle de l'analyse de traces. Les contributions présentées dans cette thèse s'inscrivent autour d'une initiative Open Source du nom de OpenTelemetry. Cette initiative vise à unifier les différentes d'approches de monitoring d'applications dans le Cloud regroupant l'analyse de journaux, la collecte de métriques et de traces, autours d'une sémantique et d'interfaces communes.

Résumé des Contributions & Présentation du Plan

Les contributions présentées dans cette thèse utilisent le projet OpenTelemetry, et en particulier la sémantique des données de télémétrie, pour présenter un modèle d'encodage de ces données sous forme de graphe de propriétés. Ce modèle d'encodage de trace est utilisé de deux façons pour illustrer deux scénarios de détection d'anomalies de performances propres aux architectures Cloud.

La thèse s'articule de la façon suivante :

A.1 : Monitoring d'Applications Cloud présente une analyse de littérature sur le paradigme Cloud et son évolution. Une attention est portée aux différentes approches de monitoring des applications distribuées ainsi qu'aux initiatives couvrant l'utilisation de systèmes de traces distribuées.

A.2 : Modélisation des Communications Internes présente la sématique OpenTelemetry et explique les mécaniques d'encodage sous forme de graphe qui ont permis de transformer des traces en un graphe de propriétés hierarchiques.

A.3 : Détection de Composition de Services Inefficaces présente la première contribution qui utilise modèle précédent pour identifier les cas où la composition de service est inefficasse au cours de l'exécution du programme. Les travaux présentés dans cette section ont été publié dans [Cassé 2021].

A.4 : Détection de Goulots d'Étranglements présente une approche basée sur l'analyse de centralité dans le graphe pour identifier les potentiels goulots d'étranglement dans une application distribuée. Les travaux de présentés dans cette section ont été publiés dans [Cassé 2022].

A.1 Monitoring d'Applications Cloud

A.1.1 Introduction

Le concept de Cloud Computing est apparu en 2008 et n'a cessé de s'imposer dans le paysage de l'informatique depuis. À l'origine, ce concept a été initié par des entreprises de l'internet telles que Amazon, elles ont tiré parti de la virtualisation et des capacités d'automatisation pour rentabiliser les ressources de leur centre de données en les louant aux usagers. À l'origine vendu comme un simple produit, ce concept a permis aux développeurs d'application d'accéder rapidement à de plus en plus de ressources de calculs à la demande. Aujourd'hui, cet accès à une puissance de calcul virtuellement infinie a permis la démocratisation d'applications nativement distribuée telle que définie dans les applications Micro-Services ou dans les pipelines de traitement Big-Data.

Aujourd'hui, les plus grandes entreprises de l'informatique ont toutes une partie de leur bizness liée aux technologies cloud : Apple et Facebook créent et utilisent leurs propres centres de données pour développer leurs services en ligne. D'un autre côté, Amazon, Google et Microsoft utilisent et revendent leurs ressources de calculs à leurs usagers. Ces derniers mettent à disposition un vaste catalogue de services allant de la location de machine virtuelle à la gestion automatisée de bases de données ; par la suite, on qualifiera ces acteurs des CSP que l'on peut traduire par Fournisseurs de Services cloud.

A.1.2 Présentation du Paradigme Cloud

Le concept de Cloud Computing est apparu, en premier lieu, dans le milieu industriel. C'est les publications [Vaquero 2008, Mell 2011] qui ont apporté une première formalisation académique. Ces publications identifient les multiples caractéristiques des plateformes Cloud qui restent majoritairement valables aujourd'hui : en particulier le fait de mettre à disposition des ressources de calcul à la demande via un catalogue (*On-demand self-service*), accessible de n'importe où sur le réseau (*Broad network access*). Ces ressources du catalogue sont fournies *as-a-Service*, séparées entre les clients, accessible quasiment immédiatement et surtout leur utilisation est quantifiée, impactant directement la facture.

Les auteurs présentent aussi trois niveaux d'abstraction pour les ressources mises à disposition par les CSPs : L'IaaS qui met à disposition des machines virtuelles ainsi que des composants réseaux virtuels, le PaaS qui met à disposition des machines préconfigurées pour exécuter du code sans se soucier de l'environnement d'exécution. Le dernier niveau d'abstraction est le SaaS qui est un logiciel avec son interface de programmation générique capable de s'adapter au besoin des multiples utilisateurs, par exemple un outil de gestion de paiements, ou bien un outil de mise en relation avec les utilisateurs.

C'est en particulier sur ce dernier aspect de niveaux d'abstraction que cette définition a été particulièrement challengée [El-Gazzar 2016, Senyo 2018]. En effet, de nombreuses technologies ne rentrant pas dans ces catégories se sont peu à peu imposées dans le paysage des technologies phares du Cloud. Le premier exemple est la containerisation qui a été démocratisée par Docker, il s'agit d'un niveau in-

termédiaire entre le IaaS et le PaaS qui offre de meilleures capacités d'isolation et portabilité des logiciels exécutés dans le Cloud. Les architectures Serverless présentées dans [Varghese 2018, van Eyk 2018] viennent aussi se glisser entre deux niveaux d'abstraction proposés dans la définition précédente. L'architecture Serverless introduit la notion de *Function-as-a-Service (FaaS)* qui se glisse entre les abstractions de PaaS et de SaaS.

Aussi, la définition actuelle fait état de centralisation au sein des centres de données, de nombreux travaux font état d'une distribution de plus en plus complexe et proche del'utilisateur afin de servir un maximum d'utilisateur avec une latence toujours plus basse. Ces travaux traitent tant de la distribution et réplication au sein des centres de données [Chaczko 2011, Unuvar 2015, Chou 2019] que du Edge Computing et de l'IoT [Khan 2019].

L'analyse de performance dans les environnements cloud est un sujet très actif dans le milieu de la recherche et donne lieu à de nombreuses publications que ce soit au travers de la vision des fournisseurs que des usagers. Les premiers cherchent à rentabiliser leur infrastructure en servant un maximum d'utilisateurs [Yu 2018, Dabbagh 2015a, Dabbagh 2015b]. De leur côté, les utilisateurs font aussi face à de nouvelles problématiques, en effet le recours aux fournisseurs tiers ne donne pas accès aux mêmes métriques pour estimer la performance logicielle d'une application. Les travaux comme [Jayathilaka 2017, Singh 2017] montrent des initiatives basées sur les niveaux de qualité de service et d'engagement par les fournisseurs. Ces travaux sont d'autant plus complexes que le milieu technologique sous-jasant est en constante évolution.

A.1.3 Les Applications Cloud Natives

Avec la démultiplication des niveaux d'abstraction permise par les technologies cloud, les développeurs d'applications ont pu adopter une approche complètement différente pour concevoir leurs logiciels. Les technologies cloud ont permis l'émergence de l'approche dite *Microservices* [Fowler 2014, Newman 2015]. Elle est vue comme l'extension de l'approche SOA, les composants sont cependant beaucoup plus découplés, il s'agit de processus indépendant communicant entre eux via le réseau, souvent en HTTP. Cette approche est même qualifiée de *fine-grained SOA* dans les travaux de [Heinrich 2017].

Les applications cloud adoptent donc une architecture nativement distribuée : ce qui permet de minimiser les temps de maintenance au point qu'il est actuellement rare de trouver des sites “en maintenance”. Les propriétés de scalabilité des systèmes distribués a permis à de nombreux sites de prendre une envergure mondiale tels que Netflix, Uber, Twitch, AirBnB ou Zoom. Par exemple dans [Gluck 2020], l'auteur, employé a Uber, mentionne que leur application se compose de plus de 2 200 microservices en production. À une telle échelle, le monitoring des composants de l'application devient une tâche extrêmement complexe, et ces entreprises se heurtent aux limites des capacités des systèmes de monitoring des applications. Dès 2016, l'entreprise Uber avait rapporté qu'elle monitorait plus de 500 000 000 séries temporelles pour s'assurer de la qualité de service de son application [Uber 2016].

À une telle échelle, les problèmes que doivent identifier les outils d'analyse de

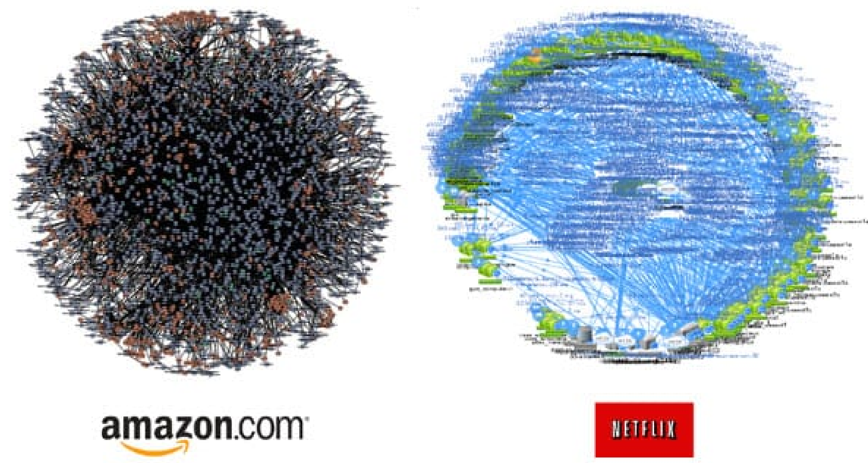

Figure A.1: Visualisation sous forme de Graphe des Microservices des applications de Amazon et Netflix en 2016 par [Cockcroft 2016a, Cockcroft 2016b].

performance ne relèvent plus uniquement de la performance individuelle de chacun des composants, mais surtout de leur composition afin de répondre aux requêtes utilisateurs. Les problèmes auxquels font face les applications de cette taille consistent à identifier des problèmes de concurrence, d'accès mutualisé aux ressources, de détection de goulots d'étranglement ou même de vampirisation d'une ressource sur une autre.

A.1.4 Monitoring et Analyse de Performance d'Application cloud

Les applications cloud sont difficiles à monitorer et ce n'est pas uniquement dû à leur taille. Elles sont faites de composants hétérogènes, de technologies appartenant à des niveaux d'abstraction différents et potentiellement hébergées sur des centres de données différents. Chaque composant émet ses propres données de performance sans possibilité de recoupement avec des données provenant d'autres sources. La masse de données créée à l'échelle d'une application tournant en production est énorme et est souvent traitée avec une approche Big-Data dans la littérature [Quantcast 2013, Dean 2014, Thalheim 2017, Dalmazo 2017, Gan 2018a, Gan 2020].

L'entreprise Facebook a publié beaucoup de travaux sur leur approche du monitoring. Au fil des années on a pu observer que l'entreprise a déployé beaucoup d'énergie à identifier la causalité entre les évènements de monitoring reportés par chacun des composants du site. Ces travaux ont commencé par la création d'un arbre de causalité basé sur les interactions entre composants [Chow 2014] pour adopter la forme d'un outil de collecte et d'analyse de traces comme on en trouve aujourd'hui [Veeraraghavan 2016, Veeraraghavan 2018, Kumar 2018]. L'orientation de ces travaux montre le besoin d'établir des regroupements logiques au sein des données de monitoring.

De nombreux travaux viennent corroborer ce besoin d'établir des regroupements entre les données émises par de multiples composants dans un système distribué

afin de retrouver une vue plus globale du système que celle de chacun de ses composants [Ardelean 2018, Lin 2018, Jayathilaka 2017]. C'est dans cette optique qu'a été créé OpenTelemetry, proposer un ensemble d'outils pour traiter les données de télémétrie des applications cloud. Le projet définit une sémantique visant à unifier le format des données tels que les Logs, les métriques, et aussi les traces des applications. De plus, le projet propose d'utiliser les techniques de propagation de contexte [Kanzhelev 2020] afin d'établir une corrélation entre les données reportées par plusieurs systèmes directement à l'exécution.

Conclusion

Les travaux de cette thèse se positionnent dans ce contexte d'adoption de la technologie OpenTelemetry comme moyen permettant la détection d'anomalies de performances inhérentes aux systèmes distribués. Ces travaux sont supportés par l'utilisation d'outils de tracing au sein de développements industriels comme ce fut le cas pour Djingo. Bien qu'actuellement le tracing des applications cloud ne soit majoritairement utilisé à des fins d'investigation, les chapitres suivants détaillent une approche capable de suivre les performances de l'application directement à l'exécution.

Dans les sections suivantes, nous détaillerons la conception d'un graphe de propriété basé sur les dépendances exprimées dans les traces. Les sections suivantes détaillent les contributions en proposant de détecter des compositions de services inefficaces et des goulots d'étranglement. Ces deux approches proposent de traiter le graphe en identifiant les cycles en en analysant la centralité des nœuds.

A.2 Modélisation des Communications Internes grâce aux Données de Traces Distribuées

A.2.1 Introduction

Le projet OpenTelemetry a démarré en mai 2019 comme union de deux projets open source existant : OpenCensus et OpenTracing. L'union de ces deux projets couvrants chacun un pan de la télémétrie dans le Cloud a permis la création d'une initiative OpenSource visant à proposer une normalisation des outils de collecte de données de monitoring. Cette initiative est encadrée par la CNCF[2] et de nombreux acteurs privés y contribuent (comme Google, Uber, Amazon, etc.). Ce projet ne vise qu'à créer un pipeline d'acheminement de données ainsi que leur format ; ni le stockage ni l'exploitation ne sont couverts par le standard.

Les travaux présentés dans cette thèse exploitent la sémantique des données de monitoring afin de modéliser une application physiquement distribuée sur plusieurs centres de données dans un graphe de propriétés hiérarchique. L'étude présentée ici s'appuie sur le cas des Cluster Kubernetes Zonaux qui sont des clusters Kubernetes distribués dans plusieurs centres de données. Alors que Kubernetes présuppose d'un réseau homogène connectant les différentes machines qui le compose, on observe que cette hypothèse est mise à l'épreuve par les récentes exploitations du système. La modélisation présentée dans cette section fait ressortir la structure hiérarchique d'un réseau, les sections suivantes proposeront deux utilisations de ce modèle.

A.2.2 Présentation de l'Écosystème de Tracing

Le projet OpenTelemetry est complexe et pour ces travaux nous nous intéresserons principalement à une sous-partie de ce projet: la sémantique des données. Le standard définit le concept de *Resource* qui caractérise l'unité depuis laquelle est émise la donnée de performance. Au sens OpenTelemetry, cette *Resource* est une seule et unique entité, caractérisant le potentiellement le Pod si l'environnement monitoré est Kubernetes. Cependant cette ressource peut aussi être une machine virtuelle, ou simplement un processus. Il est possible d'identifier le type de cette ressource en fonction des métadonnées que celle-ci possède.

Les traces OpenTelemetry se matérialisent chacune sous la forme d'un diagramme de Gantt qui représente les relevés de latence de chacune des fonctions instrumentées. Chacun de ces relevés est associé à une unique ressource et les relevés forment un Graphe Acyclique Dirigé qui représente le chainage des fonctions, par exemple associé à une action utilisateur. La figure A.2 montre un simple exemple de trace et la conversion associée en Graphe Acyclique Dirigé.

Par la suite, nous proposons une approche visant à refléter les différents niveaux d'abstraction grâce à la sémantique des métadonnées portées par ces ressources. En effet la vision actuelle de l'outil ne reflète pas la complexité structurelle des applications cloud et de leur plateforme. Dans l'approche suivante, nous ne nous

[2] *Cloud Native Computing Foundation* Organisation au même titre que la Apache foundation dédiée aux outils de Cloud. Il s'agit notamment de la fondation qui encadre le développement de projets phares tels que Kubernetes ou Prometheus.

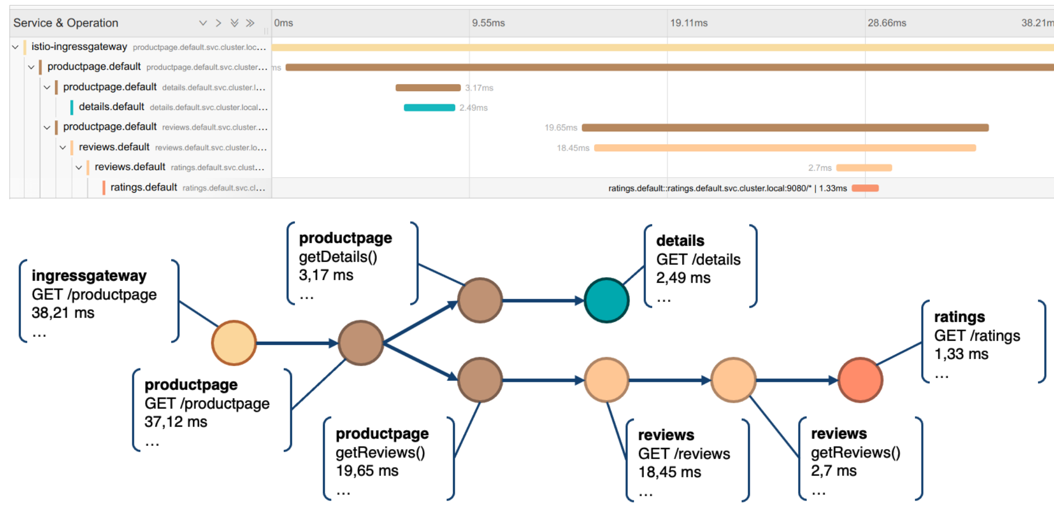

Figure A.2: Graphical Representation of Jaeger Analytic Library Graph Encoding

limiterons pas à une seule et unique ressource par relevé de latence. L'approche consiste à faire apparaitre tous les niveaux d'abstraction en tant que Noeuds distincts dans le graphe. Dans le cadre d'un cluster Kubernetes réparti sur plusieurs zones, chaque relevé de latence sera, à la fois, associé à un *Pod*[3], mais aussi à un *Node* (VM exécutant le Pod), à une zone de disponibilité (centre de données hébergeant la machine virtuelle).

A.2.3 Extraction et Aggregation des Données dans un Graphe de Propriétés

Afin de décrire le processus d'encodage de graphe capable de représenter les interactions entres les multiples ressources prenant part dans une application cloud nous présenterons en premier lieux comment ces différentes ressources sont modélisées et identifiées. En second lieu, nous présenterons un processus de réécriture de graphe afin de simplifier le graphe pour ne représenter que les ressources et leurs interactions.

A.2.3.1 Identification et Modélisation des Composants

Dans une application Cloud on peut observer que la structure des composants est extrêmement hiérarchique. Dans le cas de Kubernetes on peut observer des relations d'ordre et de précédence entre les niveaux d'abstraction que propose l'outil, par exemple $Pods \subset ReplicaSets \subset Deployments \subset Clusters.$ représente l'ordre d'inclusion logique composants. D'un autre, coté il est possible de modéliser l'ordre de positionnement physique par une hiérarchie d'inclusion différente : $Pods \subset Nodes \subset Zones \subset Clusters.$

[3] Unité logique encadrant l'exécution de containers dans un environnement Kubernetes

Dans le cas des applications physiquement distribuées, la hiérarchie de placement des ressources est particulièrement intéressante. En effet, elle permet de mettre en avant les décisions de placement prises Kubernetes. Nous avons donc décrit un processus d'encodage de traces capable d'identifier chacune de ces ressources comme un seul et unique nœud dans le graphe et chacun des relevés de latence pointera vers ce nœud. De plus, ce système d'encodage permet aussi d'associer plusieurs ressources de plusieurs types à un seul relevé de latence dans une trace.

La figure A.3 représente le rôle du processus d'encodage: il transforme un agrégat de relevés de latence (appelés *Spans*) formant diagramme de Gantt en un graphe de propriété. Ce graphe met en avant dans des nœuds dédiés les ressources exécutant l'application faisant ainsi ressortir la structure de l'application.

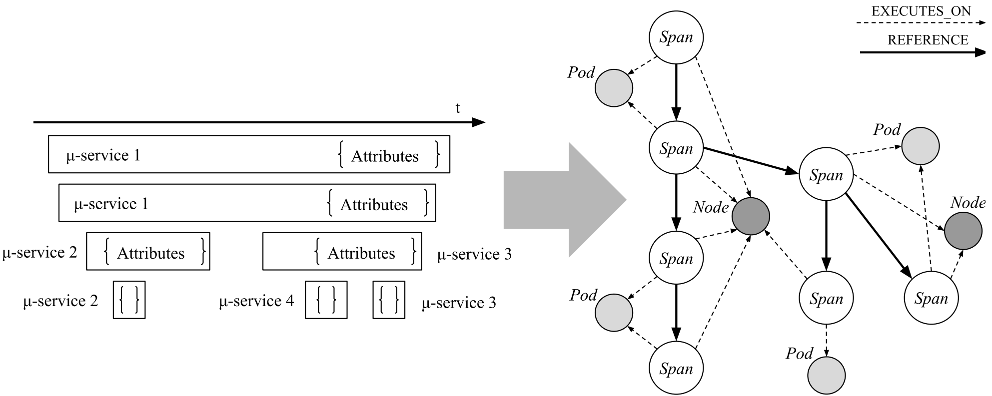

Figure A.3: Processus d'encodage d'une trace en un graphe de propriétés.

Dans cette figure nous n'avons fait apparaitre que deux niveaux de hiérarchie pour ne pas complexifier la visualisation. Avec ce graphe il est possible de voir à quel point deux relevés de latences sont "physiquement proches" dans l'application: c'est-à-dire en fonction du nombre de nœuds de type *Resource* qu'ils partagent. Cependant, ce modèle ne permet pas de mettre en avant des relations entre ces nœuds de type *Resource*. La section suivante propose une approche pour mettre en avant les relations de ces ressources avec des techniques de réécriture de graphe.

A.2.3.2 Identification et Modélisation des Interactions

Une trace traduit la composition des ressources entre elles afin de répondre à une fonction. Le graphe présenté précédemment met en avant la relation d'inclusion entre les ressources exécutant l'application. Dans le modèle présenté de la figure A.3, nous pouvons observer que les relevés de latence (*Span*) sont liés entre eux et forment un Graphe Acyclique Dirigé. Les ressources (*Pods*, *Nodes*), en revanches, ne sont pas liées entre elles. Afin de mettre en avant les relations entre les ressources, nous proposons de "projeter" les relations observées entre les *Spans* sur les nœuds représentant les différents types de ressource.

Dans une trace, les noeuds et les arrètes forment des motifs différents quand les communications représentées traversent les frontières des ressources. En effet, une dépendance entre spans observés sur la même ressource se matérialisera dans

le graphe comme deux noeuds *Span*s pointant vers le même noeud *Resource*. Au contraire, une communication réseau entre deux ressources sera représentée par deux noeuds *Span* pointant vers deux noeuds *Resource* distincts.

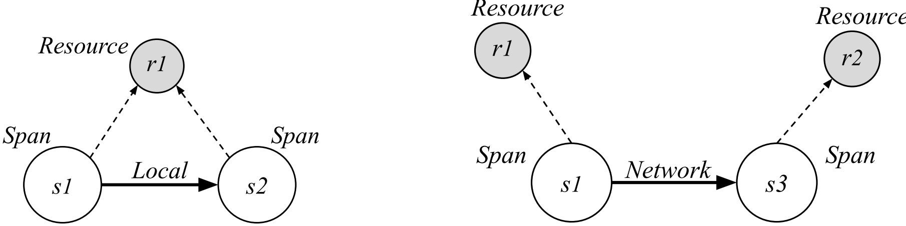

Figure A.4: Représentation des motifs de communication locale et de communication réseau dans notre modèle d'encodage de graphe.

La figure A.4 fournit une représentation visuelle de ces deux motifs. Leur identification dans le graphe peut donc permettre de représenter quelles frontières traversent les communications observées dans une application distribuée. Ce processus de projection est réalisé grâce à la réécriture de graphes, la figure A.5 fournit une formalisation de ce processus de réécriture ou on voit que seuls les motifs représentant des communications réseau sont préservés alors que ceux représentant une communication locale sont éliminés.

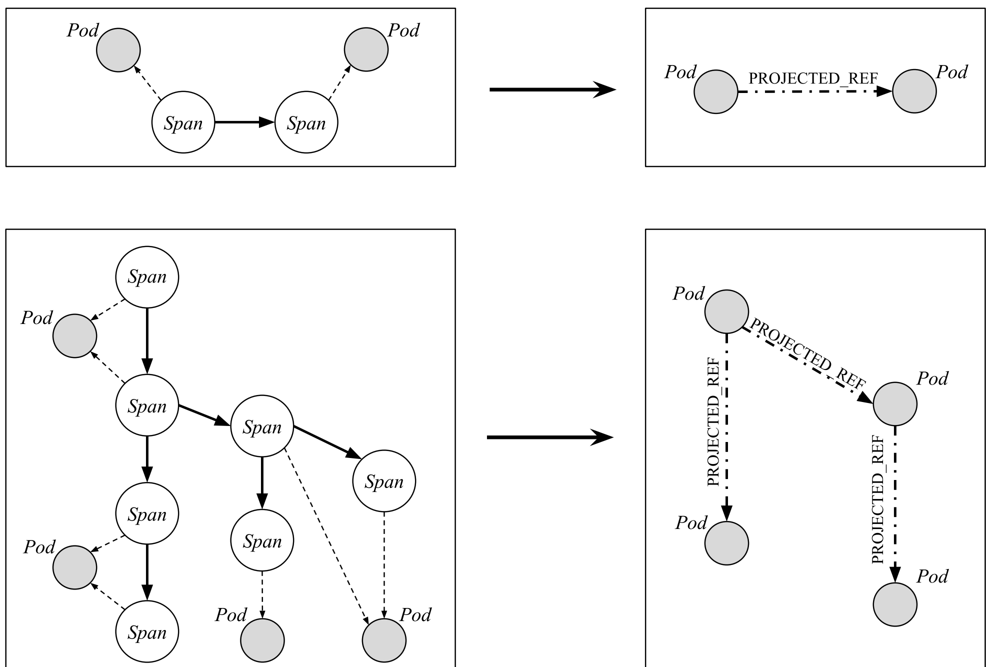

Figure A.5: Formalisation du processus de réécriture de graphe

Conclusion

Dans cette section nous avons présenté l'écosystème de tracing d'application cloud et nous avons exploité ce format pour présenter un modèle de transformation de ces données pour maintenir un graphe des communications réseau observées dans une application cloud. L'annexe B présente le code qui a été réalisé en Scala pour traiter les données de tracing et effectuer les transformations présentées. Tous les traitements présentés ont été réalisés en mémoire et parallélisés pour de meilleures performances lors de l'ingestion d'un volume plus conséquent de données.

Les sections suivantes détaillent deux applications de ce modèle pour répondre à des problématiques de performances observées sur les applications cloud notamment difficiles à adresser avec un outillage de monitoring plus conventionnel.

A.3 Détection des Communications Inefficaces dans une Application Physiquement Distribuée

A.3.1 Introduction

Dans la section précédente, nous avons vu comment les données de tracing d'application cloud pouvaient être utilisées pour maintenir un graphe de communication entre les entités prenant part à l'application. Dans cette section, nous proposons une approche pour l'analyse de ces graphes afin de mettre en avant une problématique intimement liée aux performances dans les applications physiquement distribuées. En effet, nous proposons un moyen d'identifier les cas dans lequel la composition de services peut être qualifiée d'inefficace.

Afin d'étayer notre étude, tout au long de cette section, nous nous intéresserons aux *"Cluster Kubernetes Zonaux"* qui possèdent une structure et des propriétés similaires aux infrastructures cloud étudiées dans le contexte industriel à Orange. Ces Clusters Kubernetes Zonaux correspondent à un groupement de machines réparties dans plusieurs centres de données qui contribuent ensemble à servir une application cloud. Ces clusters zonaux sont créés dans le but de répondre à des problématiques de disponibilités, mais induisent des communications supplémentaires entre les centres de données. Au sein d'Orange, la problématique soulevée par les équipes de développement d'application était d'identifier si les va et-viens entre plusieurs centres de données pouvaient avoir un impact négatif sur le temps de réponse de l'application.

A.3.2 Modélisation d'une Application cloud grâce au Concept de Hiérarchies

Le concept de hiérarchie est étudié en détail dans la publication [Zafeiris 2018]. Les auteurs présentent une définition très large du concept de hiérarchie tel qu'on peut l'observer au quotidien, puis propose une analyse plus approfondie de ce concept en identifiant des sous-catégories de hiérarchies. Parmi les sous-catégories identifiées, nous retrouvons principalement la hiérarchie d'inclusion (qui représente l'ordre d'inclusion des entités hiérarchisées comme les poupées russes ou les ressources dans une application cloud : $Pod \subset Node \subset Zone \subset Region$). L'autre sous-catégorie présentée dans la publication est la hiérarchie de flux, appelée aussi parfois hiérarchie de contrôle, elle matérialise l'influence d'une entité sur les autres entités du même type (telles que l'influence des hommes politiques ou bien la communication entre les différents microservices prenant part dans une application cloud). Bien que les auteurs détaillent d'autres types de hiérarchie dans cette publication, ces derniers ne seront pas abordés dans cette thèse.

Il apparait alors possible de modéliser différents aspects d'une application cloud grâce aux sous-catégories de hiérarchie précédentes : Une hiérarchie d'inclusion pour modéliser les différents niveaux d'abstraction des ressources et une hiérarchie de flux pour modéliser les interactions entre ces ressources. La figure A.6 représente le processus d'encodage de taces présenté dans la section précédente tout en mettant en avant ces deux types de hiérarchies pour un cluster Kubernetes classique.

Nous observons que l'ordre hierarchique entre les *Pods* et les *Nodes* se matérialise par la relation d'inclusion *IS_CONTAINED*. Les relations *REFERENCE* créant les relations *PROJECTED_REF* après réécriture du graphe forment une relation hiérarchique de flux entre ressources d'un même niveau.

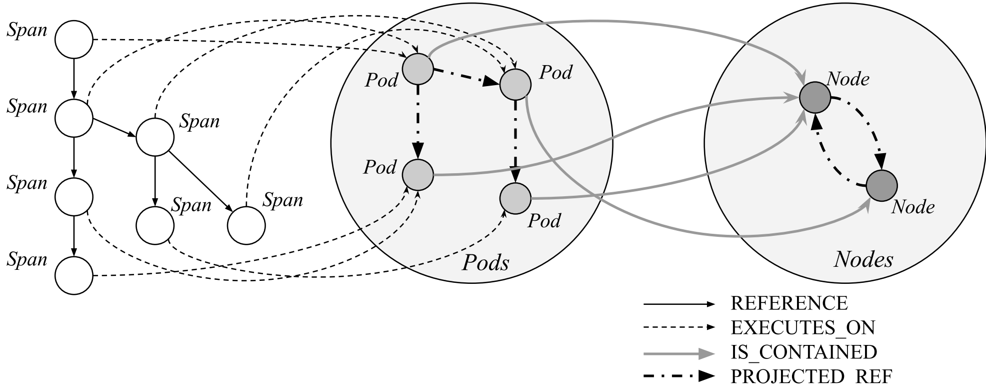

Figure A.6: Trace encodée représentée sous forme de graphe hiérarchique

Ces différentes catégories de hiérarchies ne sont cependant pas traitées dans la même mesure dans la littérature, en particulier dans le milieu informatique. Une vaste majorité des publications traite donc de la hiérarchie d'inclusion et délaisse la hiérarchie de flux, cependant, dans la publication [Luo 2011], les auteurs s'intéressent aux hiérarchies de flux et en particulier aux hiérarchies de flux imparfaites. Ils différencient les hiérarchies de flux parfaites, c'est à dire dépourvues de cycles (telles que les DAG) et les hiérarchies de flux imparfaites où la structure "ordonnée" du graphe est partiellement préservée. La présence de cycles dans un graphe interfère avec la notion d'ordre dans la hiérarchie et, de ce fait, dans leur publication, les auteurs proposent une métrique permettant de quantifier la structure hiérarchique d'un graphe. Ils présent h, l'indice de "flow hierarchy" qui quantifie à quel point un graphe possède une structure hiérarchique en dénombrant le nombre d'arêtes d'un graphe ne prenant pas part à un cycle.

Dans le modèle de trace encodées avec le méta-modèle précédent, comme celui de la figure A.6, on observe qu'un graphe initialement défini comme acyclique, pouvait, après processus de réécriture présenter des cycles dans sa structure.

A.3.3 Détection de Communications Inefficaces

A.3.3.1 Les Cycles comme Marqueurs de Communications Inefficaces

Au sein d'une application cloud, ces cycles déterminent des communications réseau entre des composants qui font des vas-et-viens. La présence de ces vas-et-viens démontre une mauvaise composition des ressources formant l'application : qu'il s'agisse d'un mauvais placement des resosurces ou d'une mauvaise décision de "routage applicatif" (comprendre équilibrage de charge).

Dans le contexte industriel fourni par Orange pour cette thèse, au cours de son développment, les concepteurs de l'application Djingo souhaitaient corréler la

présence de communication inefficaces avec le temps de réponse de l'application. La métrique de "flow hierarchy" présentée par [Luo 2011] appliqué à chacun des niveaux d'abstraction présentés dans la section précédente constitue donc un marqueur d'effiicacité des communications réseaux sans pénaliser la distribution des ressources.

A.3.3.2 Détection de Cycles

L'approche présentée dans [Luo 2011] proposait une approche pour la détection de cycles basée sur l'exponentiation successive de la matrice d'adjacence. Cette méthode se révélant extrêmement couteuse en termes de calcul, elle n'était pas adaptée à un traitement des traces en ligne.

Pour le calcul de l'indice de "flow hierarchy", j'ai proposé une approche basée sur la détection des composants fortement connectés dans un graphe couplé à un parcours de graphe. En effet, dans un graphe dirigé, les sommets appartenant au même composant fortement connectés sont nécessairement impliqués dans un cycle. La figure A.7 montre le résultat de l'identification des composants fortement connectés dans un graphe dirigé. Les sommets sont regroupés en composants fortement connectés (représentés en pointillés), pour savoir si les arêtes sont impliquées dans un cycle comme le propose [Luo 2011], pour chacune des arêtes, celle-ci n'est pas impliquée dans un cycle si les sommets qu'elles lient n'appartiennent pas au même composant fortement connecté.

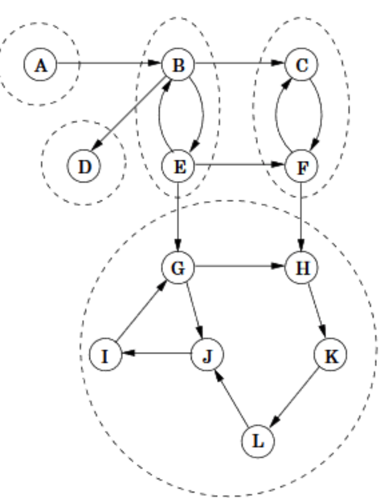

Figure A.7: Identification des composants fortement connectés dans un graphe dirigé

A.3.3.3 Calcul d'Indice de Flow Hierarchy

La formule de l'indice de "flow hierarchy" pour un graphe dirigé $G = (V, E)$ est définie telle que :

$$h = \frac{\sum_{i=1}^{L} e_i}{L} \tag{A.1}$$

Dans cette formule $L = |E|$ soit le nombre total d'arêtes dans le graphe et e_i étant un coefficient dénotant si la $i^{\text{ième}}$ arête du graphe appartient à un cycle ou non. L'indice $e_i = 0$ $i^{\text{ième}}$ arête du graphe appartient à un cycle, sinon $e_i = 1$. Ainsi on peut reformuler la formule précédente par le ratio du nombre d'arêtes **non** impliquées dans un cycle sur le nombre total d'arêtes dans le graphe. Le code présenté dans l'annexe B.4 représente les différentes étapes de ce processus.

Ainsi, sur un flux de traces émises par une application cloud, il est possible calculer, pour chacun des niveaux d'abstraction l'indice h associé. La figure A.8

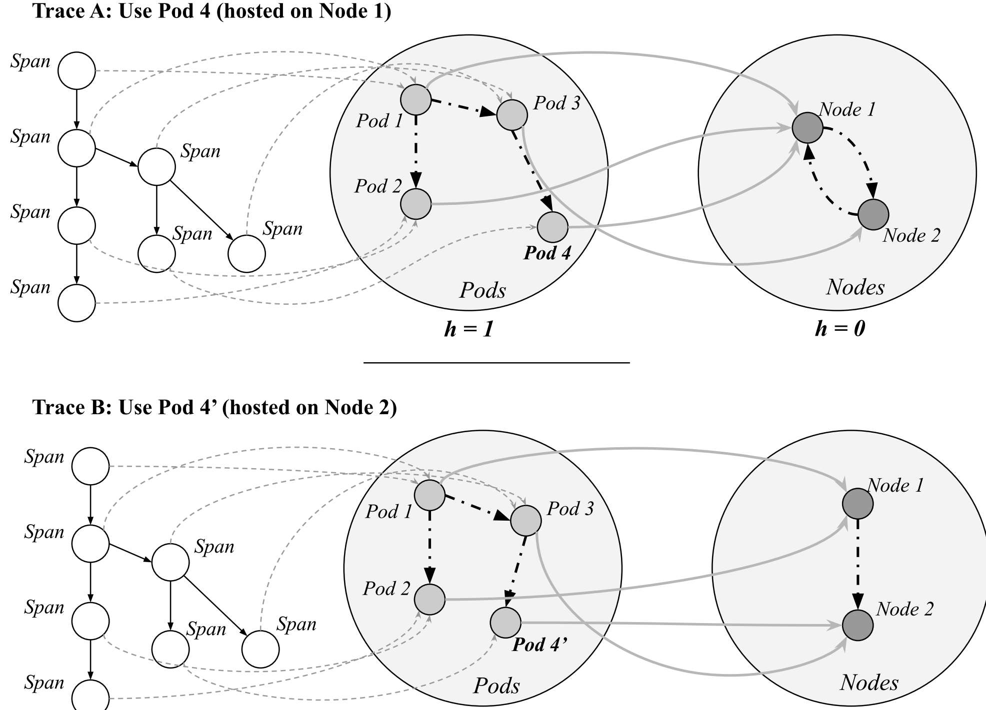

Figure A.8: Calcul d'indice de flow hierarchy sur deux placements de Pods dans un Cluster Kubernetes

montre comment, sur deux traces fictives utilisant deux instances d'un service distribué sur deux machines différentes, l'indice de flow hierarchy peut mettre en avant la présence d'un cycle.

A.3.4 Mise en œuvre

Au cours de la thèse, l'approche a été d'utiliser les données exposées par l'outil "Jaeger" qui était utilisé à Orange pour visualiser les traces de l'application *Djingo*. Cependant, au cours de la thèse, l'application Djingo a été réduite et n'utilisait plus de répartition sur plusieurs centres de données. L'approche présentée n'a pas donc pu être validée sur une application à large échelle.

La validation a été réalisée sur une maquette dont l'infrastructure se rapprochait de celle de Djingo: un cluster Kubernetes zonal. Une application de démonstration a été déployée sur un cluster zonal et celle-ci a été instrumentée pour utiliser Jaeger au même titre que Djingo. Un cluster Kubernetes zonal n'introduit pas de latence supplémentaire perceptible lors des communications entre les différentes zones, cependant ces communications entre zones sont facturées par les fournisseurs cloud.

Le but de cette maquette a donc été d'identification des communications qui allaient être facturées et surtout si celles-ci introduisaient des vas-et-viens qui allaient être facturés sans apport bénéfique pour l'application.

Conclusion

Dans cette section nous avons présenté une application du modèle présenté dans la section précédente pour un traitement online des traces afin d'identifier les placements de services introduisant des communications inefficaces. Cette problématique est propre aux applications fortement distribuées et n'est actuellement pas traitée dans la littérature actuelle. Les traces, couplées au modèle hiérarchique présenté permettent de mettre en lumière des comportements qui n'apparaissent pas dans les outils de monitoring actuels.

L'exploitation de l'application *Djingo* à permis de mettre en lumière une problématique réelle pour l'optimisation des communications dans un système distribué. Bien qu'il n'ait pas été possible de tester l'approche sur un vrai flux de données, une maquette a montré la validité de l'approche.

A.4 Détection de Goulots d'Étranglements dans une Application Cloud

A.4.1 Introduction

Dans cette section, nous nous intéresserons à une autre application du modèle d'encodage de graphe présenté dans la section A.2. Au lieu d'appliquer une approche traitant chaque trace individuellement au runtime, nous proposons dans cette partie d'exploiter l'accumulation des traces dans un graphe de communication hiérarchique. Ainsi, dans cette section nous exploiterons le modèle d'encodage de trace afin d'identifier les potentiels goulots d'étranglement dans une application cloud distribuée.

Cette étude sera réalisée grâce à un simulateur d'applications cloud qui a été modifié pour émettre des traces au format OpenTelemetry. Le simulateur utilisé s'appelle *Spigo* et est disponible sur GitHub[4], son auteur, A. Cockcroft, a conçu ce simulateur pour représenter la figure A.1 présentée dans la première section. Ce simulateur a permis de modéliser diverses applications cloud suivant les pratiques d'architectures proposées par AWS, en particulier sur le positionnement des ressources dans diverses régions sur le globe.

Ces travaux s'inscrivent dans le cadre des applications maintenues par Orange dans lesquelles l'identification de goulot d'étranglement est critique pour assurer et maintenir le service. Cette section illustre donc l'impact d'une instrumentation de tracing avec OpenTelemetry pour identifier, les services les plus risqués au sein d'une application distribuée.

A.4.2 Généralisation de l'Encodage en Graphe de Propriétés

Jusqu'à présent, les travaux d'encodage et d'exploitation de graphe de propriété étaient intimement liés aux clusters Kubernetes zonaux. Cependant, la sémantique des ressources présentées dans le projet OpenTelemetry couvre bien plus de types de ressources, et inclut les applications AWS. Ainsi, nous avons adapté le processus d'encodage de graphe afin qu'il supporte plus de ressources pour modéliser les applications AWS au même titre que les applications Kubernetes.

L'adaptation de ce processus couvre à la fois la modélisation des relations `EXECUTES_ON` que l'on observe entre les *Spans* et les *Resources* ainsi que la modélisation des relations `IS_CONTAINED` qui traduit la hiérarchie d'inclusion des *Resources* entre elles. La figure A.9 représente la décomposition en deux étapes pour la découverte du méta modèle d'encodage de traces. La partie gauche de ce méta modèle présente capable l'extraction de multiples relations `EXECUTES_ON` d'une trace en suivant la sémantique décrite dans le projet OpenTelemetry.

La partie droite, quant à elle, représente l'ajout des relations `IS_CONTAINED` qui n'apparait pas explicitement dans les données de tracing. En effet, cette seconde partie nécessite une connaissance préalable de l'application et de sa structure. Ces relations `IS_CONTAINED` représentent l'inclusion des différents types de ressources entre eux, et ne sont pas tributaires des données captées au runtime. Ainsi, il est

[4] `https://github.com/adrianco/spigo`

possible d'ajouter, par réécriture de graphe, la relation d'inclusion $Pod \subset Node \subset Zone \subset Region$ dans un deuxième temps comme représenté dans la figure suivante. L'ordre d'inclusion des ressources reste un paramètre passé au modèle d'encodage de trace précédent, seuls les sommets de type ressource liés entrent eux par une relation `IS_CONTAINED` sont préservées au cours de la réécriture de graphe.

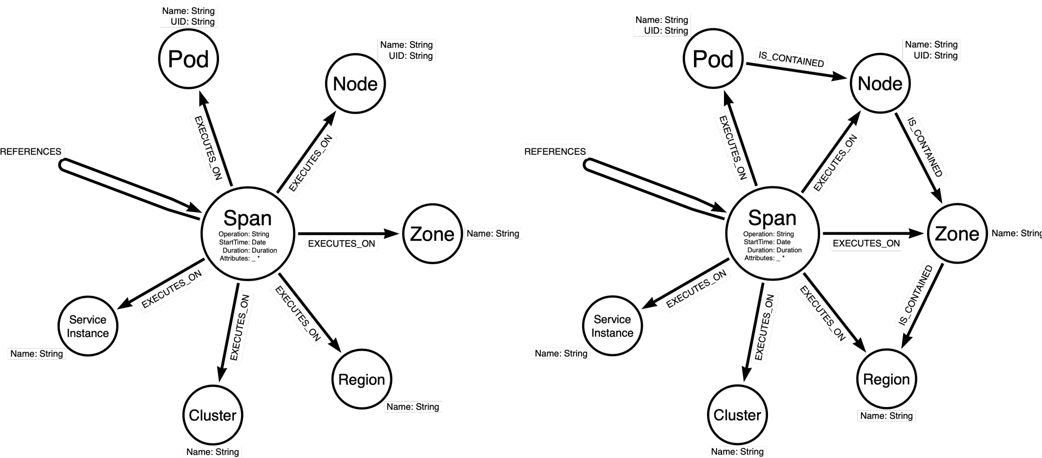

Figure A.9: Décomposition du modèle d'encodage de graphe en deux étapes pour les relations *EXECUTES_ON* et *IS_CONTAINED*.

Dans le cas des applications AWS, l'ordre d'inclusion des ressources est le suivant : $Service \subset Zone \subset Region$. Toutes ces valeurs sont émises dans les données de tracing et peuvent donc être aisément extraites et traitées par le processus d'encodage. La figure A.10 représente le méta modèle qui sera utilisé pour analyser les applications AWS modélisées par le simulateur *Spigo*.

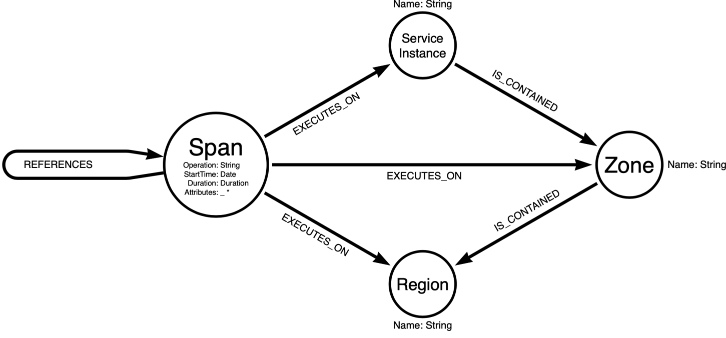

Figure A.10: Méta Modèle du processus d'encodage de graphe après les étapes d'extraction et de sélection des relations.

A.4.3 Utilisation de l'analyse de Centralité pour l'Anticipation de Goulots d'Étranglement

Grâce aux données de tracing et aux méthodes d'encodage présentées dans cette thèse, il est possible de créer et de maintenir un graphe de communication des différents types de ressources impliquées dans une application microservices. Dans cette section nous proposons d'utiliser des algorithmes de calcul de centralité dans un graphe pour identifier les goulots d'étranglement dans une application Cloud physiquement distribuée. En effet, dans un graphe, l'analyse de centralité permet d'identifier les sommets les plus importants. Il existe plusieurs algorithmes de calcul de centralité, chacun proposant une définition différente de *l'importance* accordée à chacun des sommets étudiés.

La figure A.11 représente un graphe mettant en avant différentes topologies dans lequel certains sommets sont identifiés par une lettre, chacune de ces lettres représente le somment le plus important selon un algorithme qui sera présenté par la suite.

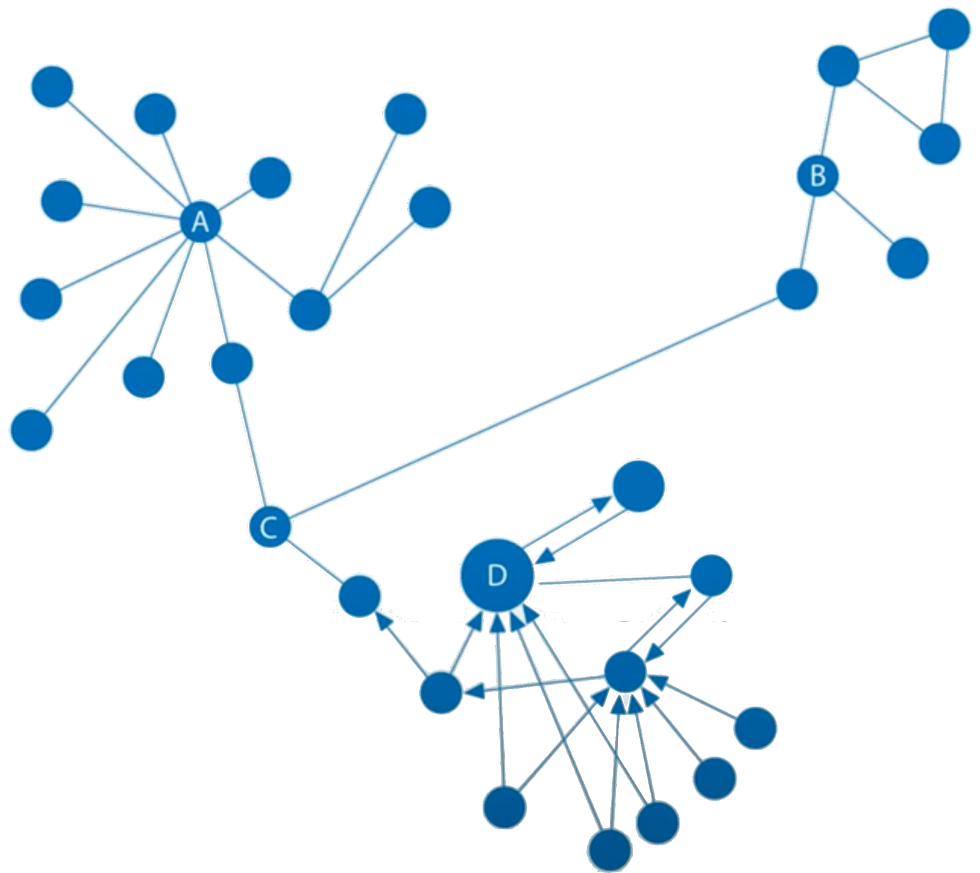

Figure A.11: Graphe mettant en avant différentes topologies pour illustrer les différents algorithmes de centralité.

Le sommet A: **Il possède le plus haut score de centralité de degré:** la centralité de degré mesure l'importance d'un sommet par le nombre de connexions qu'il possède avec le reste du graphe. On peut identifier deux types de centralités de degré, la centralité de degrés entrants et la centralité de degré sortant qui s'applique pour les graphes dirigés. La première compte uniquement les relations entrantes la seconde uniquement les sortantes. Pour un graphe $G = (V, E)$, la centralité de degré peut être exprimée pour chaque sommet $u \in V$ grâce à la formule suivante :

$$\mathcal{C}_d(u) = k_u$$

Le sommet B: **Il possède le plus haut score de centralité de proximité:** la centralité de proximité accorde une forte importance aux sommets les plus proches de l'ensemble des autres sommets du graphe. Pour un graphe $G = (V, E)$, la centralité de proximité peut être exprimée pour chaque sommet $u \in V$ grâce à la formule suivante :

$$\mathcal{C}_c(u) = \frac{|V| - 1}{\sum_{\substack{\forall v \in V \\ u \neq v}} d(u, v)}$$

Le sommet C: **Il possède le plus haut score de centralité d'intermédiarité:** la centralité d'intermédiarité accorde une forte importance aux sommets agissant comme point de passage lors de l'identification d'un plus court passage entre deux sommets. La centralité intermédiaire permet d'identifier les sommets drainant le plus de trafic dans un réseau de communication notamment. Pour un graphe $G = (V, E)$, la centralité d'intermédiarité peut être exprimée pour chaque sommet $u \in V$ grâce à la formule suivante :

$$\mathcal{C}_b(u) = \sum_{s \neq t \neq u} \frac{\sigma_{st}(u)}{\sigma_{st}}$$

Dans cette équation, σ_{st} représente le nombre total de plus courts chemins quelque soit les sommets $(s, t) \in T^2$ et $\sigma_{st}(u)$ représente le nombre de plus courts chemins passant par u.

Le sommet D: **Il possède le plus haut score de centralité PageRank:** L'algorithme PageRank est une variante de l'algorithme d'une centralité de vecteur propre qui a été publié par [Brin 1998] afin supporter le fonctionnement du moteur de recherche de Google. Les algorithmes de centralité de vecteur propre estiment l'importance d'un sommet basé sur l'importance des sommets connexes. Pour un graphe $G = (V, E)$ avec λ étant la valeur propre de la matrice d'adjacence du graphe G, la centralité de vecteur propre peut être exprimée pour chaque sommet $u \in V$ grâce à la formule suivante :

$$\mathcal{C}_e(u) = \frac{1}{\lambda} \sum_{t \in M(u)} \mathcal{C}_e(t)$$

Parmi tous ces algorithmes de centralité, on peut identifier notamment *la centralité de degré entrante* et *la centralité d'intermédiarité* comme étant directement

applicables à un réseau de communication entre microservices. En effet, dans la littérature actuelle qui traitre de l'identification de goulots d'étranglement, on distingue deux typologies de goulots d'étranglement, un premier de *saturation* et un second de *concurrence* [Ibidunmoye 2015, Marvasti 2013, Veeraraghavan 2016, Veeraraghavan 2018].

La centralité de degré entrant peut être appliquée pour identifier la *saturation* de certaines ressources, en effet des ressources saturées sont trop sollicitées et atteignent les limites des ressources qui leur sont allouées. Les ressources saturées sont identifiées par un manque de CPU ou de mémoire ou d'accès disque, l'identification de ressources saturées relève du monitoring de chacun des systèmes individuellement et ne nécessite pas une vision globale.

Au contraire, les goulots d'étranglement de *concurrence* ne manifestent pas d'anomalie de performance eux même, cependant ils sont souvent limités par des dispositifs logiciels tels que des sémaphores, des accès partagé, ou des files de messages faisant office de ressource limitante qui doit être partagée. L'identification de systèmes drainant une forte quantité de trafic dans une application permettrait de donner une vision précise sur les services les plus critiques. L'utilisation de l'algorithme de centralité d'intermédiarité propose une opportunité pour détecter les composants critiques au bon fonctionnement de l'application.

La section suivante propose une approche expérimentale basée sur le simulateurs d'application AWS *Spigo* présenté en introduction, nous y présenterons le mode opératoire qui consite à valider cette approche.

A.4.4 Vérification Expérimentale

Pour valider l'identification de composants critiques, nous avons utilisé un simulateur sur lequel nous avons exécuté différentes scénarios en faisant varier le nombre d'instances de certains services pour regarder l'impact sur l'indice de centralité intermédiaire. Le simulateur *Spigo* implémente divers scénarios, simulant des application Cloud créées pour être exécutées par AWS. Parmi ces scénario, un s'intitule *Riak* qui modélise une chaîne d'acquisition de données dans un Cloud provenant de diverses zones.

Le simulateur fonctionne en modélisant chaque service d'une application AWS par une routine dédiée sur la machine. Chacune de ces routine possède son propre cycle de vie et les communications entre ces routines sont modéliées par un échange de messages entre ces routines. Le simulateur a donc été modifié pour instrumenter les échanges de messages comme si elles étaient des communications réseaux et associer a chacune des routine sa propre hiérarchie de ressources.

La figure A.12 représente le chaînage logique des composants AWS entre eux pour modéliser une application similaire à l'outil du commerce *Riak*[5], il est important de noter que ces composants sont répliqués sur différents centres de données qui n'apparaissent pas sur ce schéma.

Dans ce scénario, les composants *analystics*, *ingest*, *enricher*, *stream* et *normalization* sont des composants sans état scalables à convenance. D'un autre côté, les composants *enrichMQ* et *ingestMQ* sont des files de messages, c'est-à-dire des

[5] `https://riak.com/`

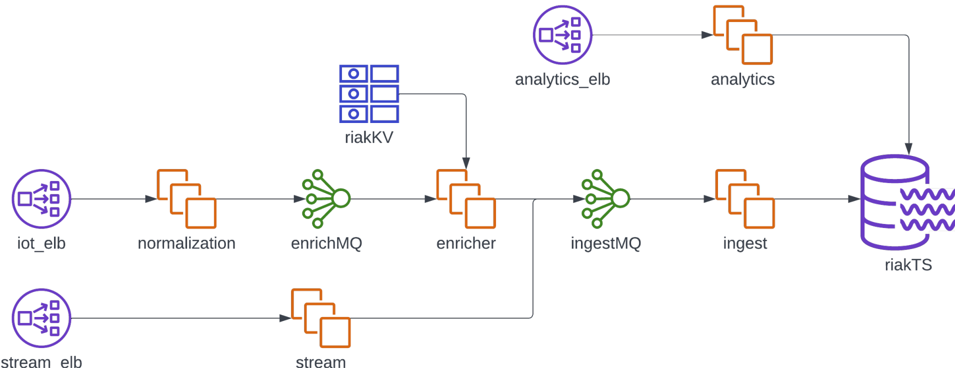

Figure A.12: Architecture logique des composants formant le scénario *Riak* du Simulateur Spigo

dispositifs complexes qui jouent un rôle primordial dans l'application. Les files de messages sont des services non triviaux à passer à l'échelle dynamiquement, leur scaling peut prendre du temps dans un système traitant un volume de données utilisateur. Dans ce cas, *ingestMQ* occupe un rôle prépondérant dans l'application, en effet cette file de messages est un point de passage obligatoire pour les données collectées par l'outil.

L'exécution du scénario dans *Spigo* nécessite une configuration décrivant le placement des services dans les différentes zones géographiques. Ainsi, à moins d'une précision contraire, chacun des services présentés dans la figure A.12 est redondé trois fois dans chaque zone de disponibilité et il y a trois zones de disponibilités. L'exécution d'une telle simulation crée des traces qui sont encodées comme présentées dans les sections précédentes. Chacune des traces est ajoutée à une base de données orientée graphe *Neo4J* et le processus de réécriture de graphe nous permet d'isoler les différentes ressources.

Il est maintenant possible de valider notre approche en applicant un algorithme de centralité d'intermédiarité sur le graphe de services. La figure A.13 représente ce graphe de service dans lequel les sommets ont une taille proportionnelle à l'indice de centralité calculé par l'algorithme. Les neuf sommets cerclés de bleu dans cette figure sont les sommets représentant les neuf instances du service *ingestMQ*. Il s'agit aussi des sommets avec le plus fort score de centralité intermédiaire, compris dans l'intervalle $[181, 193]$ alors que tous les autres services ont un score inférieur à 105. L'algorithme de centralité d'intermédiarité permet donc de bien mettre en avant les services que nous avions identifiés comme critiques après une analyse manuelle de l'architecture de l'application.

Afin de valider cette approche, nous avons modifié la configuration de la simulation pour changer le nombre d'instances du service *ingestMQ*. Dans une première simulation, nous avons descendu le nombre d'instances de *ingestMQ* à une seule instance par zone de disponibilité, rendant ce service encore plus critique. Dans un second scénario, nous avons augmenté le nombre d'instances de ce service à cinq par

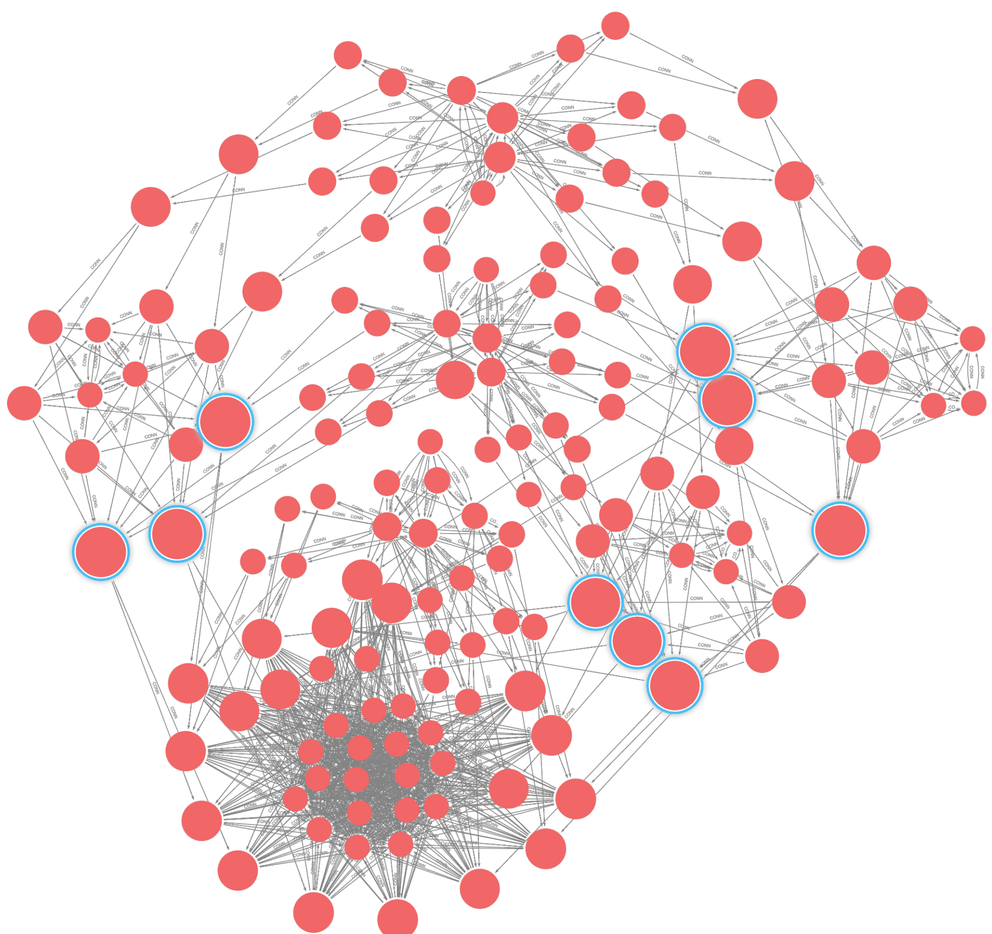

Figure A.13: Visualisation du scénario *Riak* de *Spigo* avec trois instances de *ingestMQ* par zones dans laquelle sommets du graphe sont d'autant plus gros que leur indice de centralité intermédiaire est élevé.

zones de disponibilité, ceci rendant le service moins risqué, et donc moins critique. La figure A.14 représente ces deux scénarios, on y voit toujours les sommets du graphe représentant les instances des services *ingestMQ* (cerclés en bleu quand la taille du sommet ne permettait pas d'afficher le nom). À l'instar de la figure A.13, la taille des sommets est proportionnelle au score de centralité intermédiaire calculé pour sur ce sommet. Il apparait effectivement que le score de centralité intermédiaire basé sur le graphe de trace permet d'identifier les services les plus critiques. Le tableau A.1 représente le score de centralité d'intermédiarité pour chacun des services dans l'application cloud simulée de *Riak*, chacune des colonnes notées $\mathcal{C}_{bn}$ représente les scores de centralités pour le service dans la simulation contenant n instances du service *ingestMQ*.

Il apparait que le score de centralité intermédiaire décroit d'autant plus que le service *ingestMQ* est répliqué. Ce score de centralité d'intermédiarité permet d'identifier les services les plus utilisés dans l'application et donc d'identifier de po-

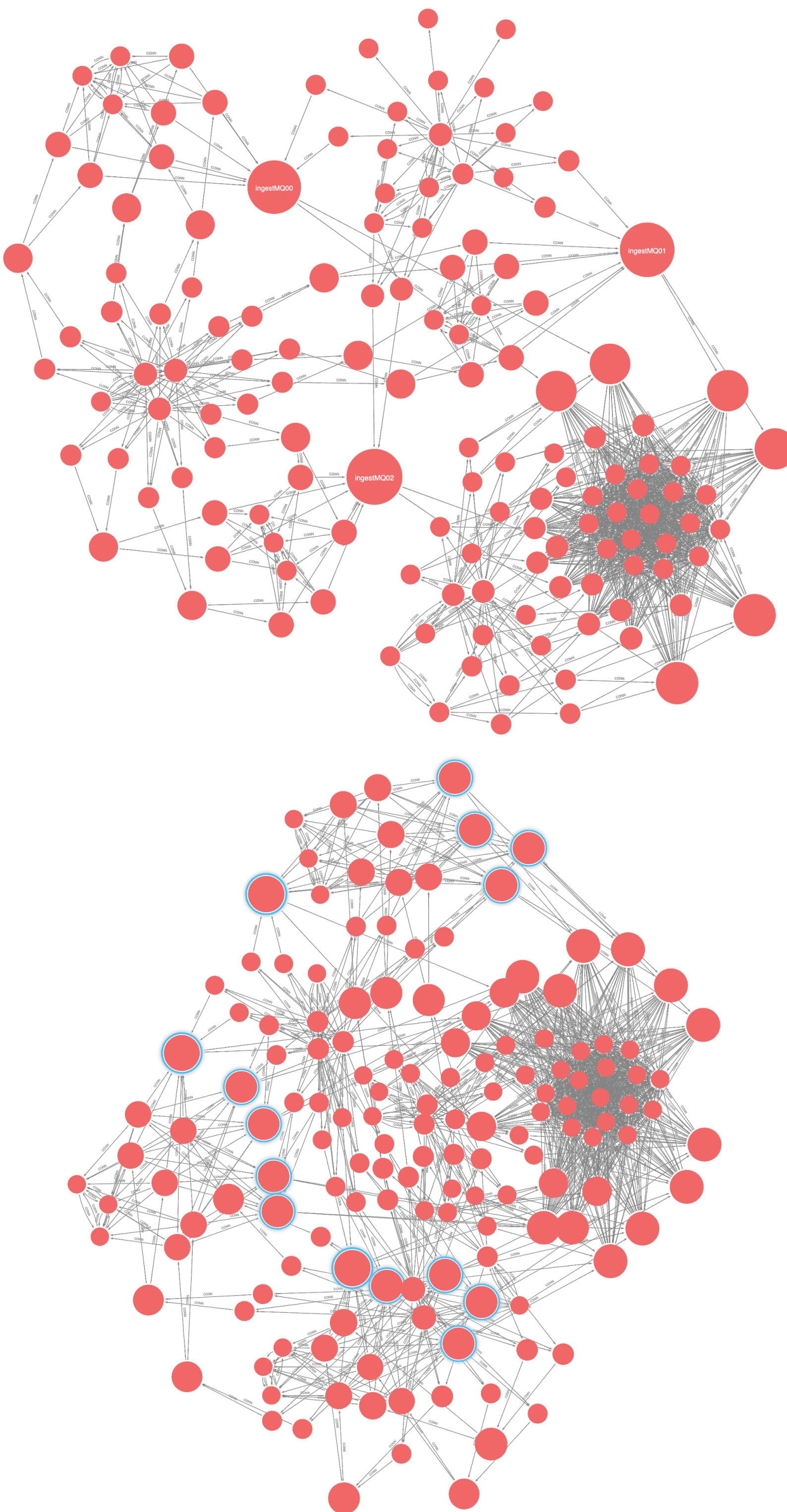

Figure A.14: Visualisation des graphes après calcul de l'indice de centralité d'intermédiarité pour les cas où le nombre d'instances de *ingestMQ* est respectivement de 1 et de 5 par zones de disponibilités

Service Name	$\mathcal{C}_{b1}$	$\mathcal{C}_{b3}$	$\mathcal{C}_{b5}$
`ingestMQ`	$[383, 422]$	$[181, 193]$	$[104, 151]$
`ingester`	$[187, 219]$	$[97, 104]$	$[118, 123]$
`enrichMQ`	$[74, 77]$	$[87, 97]$	$[92, 110]$
`enricher`	$[43, 44]$	$[53, 57]$	$[57, 65]$
`normalization`	$[14, 16]$	$[13, 18]$	$[13, 25]$
`stream`	$[3, 7]$	$[3, 12]$	$[8, 12]$
`analytics`	$[5, 10]$	$[5, 9]$	$[5, 9]$
`riakTS`	$[0, 0]$	$[0, 0]$	$[0, 0]$
`riakKV`	$[0, 0]$	$[0, 0]$	$[0, 0]$

Table A.1: Score de centralité intermédiaire associé à chacun des services de l'application cloud simulée *Riak* pour les cas ou le nombre d'instances du service *ingestMQ* est respectivement de 1, 3, et de 5.

tentiels goulots d'étranglement que les opérateurs de l'application devront surveiller avec une attention particulière.

Conclusion

Dans cette section, nous avons présenté une approche alternative au traitement du modèle de propriétés hiérarchique créé par les traces d'applications. Au lieu de traiter individuellement chaque trace, cette approche traite le graphe complet afin d'identifier les services qui auraient le plus d'impact en cas de panne de l'application ou bien les services les plus susceptibles de devenir des goulots d'étranglement. Obtenir ces informations sans données de tracing ni modèles de performance est actuellement une tâche difficile qui est industrialisée sur certaines plateformes très précises. La solution présentée ici à l'avantage d'être générique et s'adapte à toutes les hiérarchies d'inclusion de ressources.

Ces travaux sur l'étude de la centralité dans les graphes représentant des applications Cloud ont été publiés dans la conférence ICOIN2022 [Cassé 2022]. En revanche, il n'a pas été possible de procéder à une étude numérique, car les latences observées sur les communications entre processus ne suivent pas les mêmes lois stochastiques que les communications observées entre microservices potentiellement hébergés sur des datacenters différents.

A.5 Conclusion

Au cours de ces dernières décennies, les technologies cloud ont eu un impact considérable sur les applications Clouds, ils ont permis de démocratiser les architectures fortement distribuées, résilientes et dynamiques. Le monitoring de ces applications a, lui aussi, été considérablement affecté par l'adoption de technologies cloud et l'écosystème de l'analyse de performance a été impacté à son tour. Au cours de cette thèse, nous avons vu l'apparition d'un nouveau standard nommé OpenTelemetry qui a eu pour approche d'unifier les différentes initiatives concernant le monitoring des applications cloud.

Cette thèse s'inscrit dans l'adoption de OpenTelemetry chez les développeurs d'application cloud en proposant une utilisation des données de tracing pour la résolution de problèmes inhérents aux applications de nouvelle génération. Les problématiques adressées couvrent la distribution des services qui composent une application au sein de multiples centres de données, il s'agit d'un sujet de recherche actif dans la littérature. Cette thèse propose une nouvelle utilisation des données de tracing afin de traiter cette problématique sous un nouvel angle.

A.5.1 Synthèse des Contributions

Dans cette thèse nous avons abordé les sujets suivants :

- Une revue de la littérature axée sur l'environnement cloud abordant à la fois les défis architecturaux les défis d'observabilité des applications cloud. Cette étude a souligné la nécessité d'un modèle de Cloud Computing plus évolutif ainsi que l'impact de la structure distribuée pour l'analyse de performance, en particulier à grande échelle.

- La définition d'un modèle d'encodage des traces *OpenTelemetry* qui permet de reconstituer une vision globale d'une application distribuée en se concentrant sur les interactions entre ses composants. Ce modèle s'est avéré être suffisamment adaptatif pour s'adapter à plusieurs architectures cloud.

- Une proposition d'utilisation de ce modèle dans une application physiquement distribuée pour identifier les cycles pouvant induire une latence et un cout supplémentaire lors de la composition de services à partir de plusieurs zones dans des clusters zonaux Kubernetes. Cette étude s'est basée sur le besoin exprimé par l'équipe d'Orange en charge du développement de *Djingo* (un assistant vocal) d'avoir une meilleure observabilité sur la composition des services inefficaces et leur corrélation avec les performances globales des applications.

- Une autre proposition d'utilisation de ce modèle pour repérer les goulots d'étranglement grâce à un algorithme de centralité appliqué dans un graphe de services. Au lieu de traiter chaque trace individuellement, cette proposition considère l'accumulation de traces pour recréer un graphe d'interactions de microservices. Un environnement de simulation a été utilisé pour recréer une application à grande échelle, ce programme utilise des threads et des files

d'attente de messages pour modéliser les *micro services* et les *appels réseau*. Cette simulation démontre l'utilisation de l'algorithme de centralité intermédiaire pour détecter automatiquement les points d'étranglement dans un graphe de microservices.

Les propositions effectuées dans cette thèse ont pour but de proposer une utilisation complémentaire des traces d'applications distribuées à leur utilisation habituelle (pour l'investigation lors du développement). Afin de pouvoir être utilisées dans un environnement industriel, le choix a été fait de développer de s'interfacer avec le plus d'outils standardisés et open sources : nous nous sommes intéressés au format OpenTelemetry, à l'outil *Jaeger* pour accéder et visualiser les traces. De plus les traitements d'encodage de graphe ont été codés en Scala et supportent la parallélisation pour traiter plus rapidement un volume conséquent de données.

A.5.2 Pistes de Poursuite des Travaux

La majorité des travaux présentés dans cette thèse portent sur une preuve de concept appliquée sur des maquettes vaguement représentatives de différents environnements de déploiement cloud. Ainsi ces travaux n'ont pas pu être renforcés par des études numériques, la représentativité des maquettes ne permettant pas de modéliser tous les comportements des applications cloud actuelles. Le standard OpenTelemetry a gagné en maturité le temps de réaliser ces travaux et son intégration dans une application en production semble aujourd'hui envisageable.

Dans un court terme, il serait possible d'intégrer de concept de hiérarchie d'inclusion présentée dans la section sur le modèle dans des outils tels que *Jaeger*. En effet, ces outils ne fournissent pas de vue globale de l'application et permettent simplement de parcourir les traces collectées par l'outil. Le modèle, présenté sous forme de graphe hiérarchique, pourrait fournir une vue de plus hauts niveaux aux opérateurs de l'application cloud, notamment grâce à une visualisation telle que l'*Edge Bundling* comme présentée en figure A.15. Cette visualisation met au premier plan les communications interservices dans une visualisation radiale, plus une communication observée entre deux services passe proche du centre, plus elle traverse des niveaux hiérarchiques.

À moyen terme, quand les données de tracing seront plus démocratisées dans le monitoring d'application cloud, il serait possible d'affiner les métriques proposées dans cette thèse :

- Pour l'indice de *flow hierarchy*, une métrique complémentaire qui permettrait d'identifier les cas où les allées retours sont très denses, et donc où la composition de service est très inefficace, serait le nombre d'arêtes minimum à ôter au graphe afin qu'il soit dépourvu de cycles.

- Pour la détection de goulots d'étranglement à l'aide d'un calcul de centralité d'intermédiarité, un calcul plus représentatif serait, non pas de calculer le plus court chemin entre toutes les paires de sommets du graphe, mais d'utiliser les passages observés et représentés par les traces.

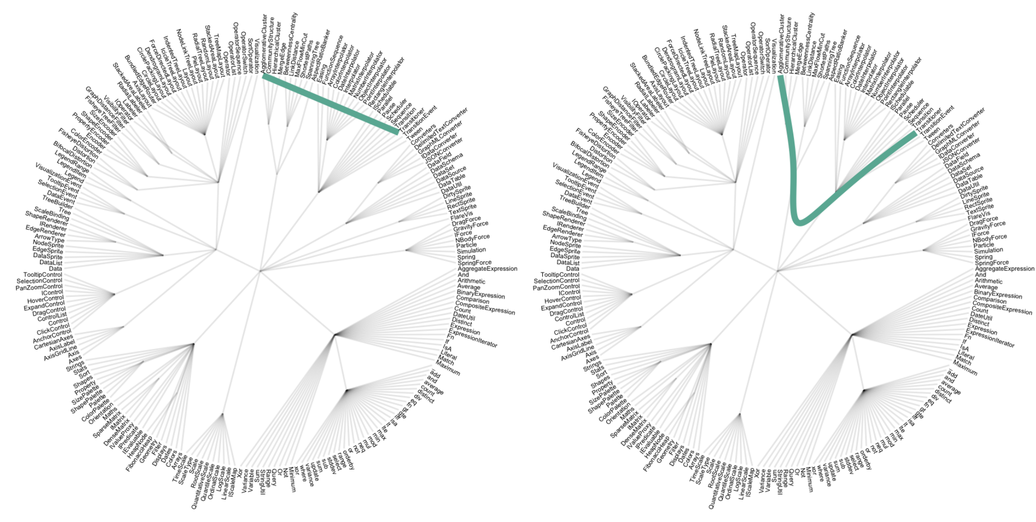

Figure A.15: Visualisation *Edge Bundling* matérialisant la hiérarchie d'inclusion par une tension des arcs plus forte vers le centre quand le nombre de couches hiérarchiques traversées est élevé.

Enfin a plus long terme, on pourrait voir les mécanismes permettant de forger les traces adoptées à d'autres environnements fortement distribués tels que les systèmes d'information d'entreprises, l'IoT ou les infrastructures 5G. Dans ces infrastructures, où la notion de confiance est primordiale pour assurer la sécurité des échanges, l'utilisation d'un graphe de communication entre les différentes parties prenantes d'un tel système pourrait aider à identifier les comportements malveillants. Le modèle présenté nécessiterait des adaptations pour s'adapter à un nouvel écosystème, mais l'observation des communications au sein du système ouvre des portes à l'identification de comportements anormaux grâce aux différentes techniques permises par l'analyse de graphes.

Appendix B

Scala Notebook and Code

Contents

B.1 Data Aquisition **141**
B.1.1 Establishing a Channel With a Jaeger Instance 141
B.1.2 Mapping ProtoBuf Data to Standard Java/Scala API 143
B.2 Definition of the Analytics Trace-Data-Model **146**
B.2.1 Operation Entities 146
B.2.2 Resources Entities 146
B.2.3 Span Entities 147
B.2.4 Trace Entities 148
B.3 Graph Encoding **149**
B.3.1 Defining the Property Graph Model 149
B.3.2 Encoding Process 150
B.3.3 Graph Rewriting 152
B.4 Calculation of the Flow hierarchy metric **153**
B.5 Building and Running the Pipeline **155**

Introduction

This appendix presents the Polynote Scala notebook implementing and executing the model and the computation of the flow hierarchy metric.

B.1 Data Aquisition

B.1.1 Establishing a Channel With a Jaeger Instance

To establish a connection with the gRPC API v2 of the Jaeger instance we define a `JaegerAPIClient` class that will handle the query, and aggregation of Span data into instances of the model we defined. In this code, we create a gRPC channel with Jaeger API v2 and create a Trace Request on the root service that will be run in the rest of the notebook.

To process Spans instances into traces a method `batchAggregateAs[A]` has been defined: The method applies a "process" function on a batch of Jaeger Spans. The process function take all spans as input and shall output a collection of elements typed `Either[Iterable[ProtoSpan], A]`:

- The `Right` part of the process function returned type are individual elements typed `A` returned by the `batchAggregateAs` method.
- The `Left` part is a collection of Spans that have not been processed but that shall be processed in the future (e.g. spans of unfinished traces). These spans are saved into the `JaegerAPIClient.orphanSpans` list and will be appended to the next collection of Jaeger Spans retrived by the `batchAggregateAs` function.

`JaegerAPIClient` can be used to build traces out of Spans: either traces can be built from spans, either the trace is not complete and building this trace is defered to the next batch.

```
import com.google.protobuf.ByteString
import io.grpc.{ManagedChannel, ManagedChannelBuilder}
import io.jaegertracing.api_v2.Model.{Span => ProtoSpan}
import io.jaegertracing.api_v2.Query._
import io.jaegertracing.api_v2.QueryServiceGrpc
import
↪ io.jaegertracing.api_v2.QueryServiceGrpc.QueryServiceBlockingStub

import collection.JavaConverters._

class JaegerAPIClient(jaegerQueryHostPort: String) {
  // Allow to keep some spans between two batch function like
  ↪ batchAggregateAs
  private var orphanSpans: List[ProtoSpan] = List[ProtoSpan]()

  private val channel: ManagedChannel = ManagedChannelBuilder
    .forTarget(jaegerQueryHostPort)
    .usePlaintext
    .build

  val queryService: QueryServiceBlockingStub =
  ↪ QueryServiceGrpc.newBlockingStub(channel)

  def queryTraces(serviceName: String, operationName: Option[String]
  ↪ = None): Iterator[ProtoSpan] = {
    val queryBuilder = operationName match {
      case Some(op) =>
      ↪ TraceQueryParameters.newBuilder.setServiceName(serviceName).setOperationName(o
      case None      =>
      ↪ TraceQueryParameters.newBuilder.setServiceName(serviceName)
    }
    val query = queryBuilder.build

    val jaegerTraceRequest =
    ↪ FindTracesRequest.newBuilder.setQuery(query).build
```

```
    for {
      spanChunks <-
       ↪  queryService.findTraces(jaegerTraceRequest).asScala
      protoSpan  <- spanChunks.getSpansList.asScala
    } yield protoSpan
  }

  def batchAggregateAs[A](batchSize: Int, serviceName: String,
   ↪  operationName: Option[String] = None,
                          process: Iterable[ProtoSpan] =>
                           ↪  Iterable[Either[Iterable[ProtoSpan],
                           ↪  A]]): Iterable[A] = {
    println(s"Retrieving ${batchSize} spans from Jaeger API")
    val newSpanBatch = this.queryTraces(serviceName,
     ↪  operationName).take(batchSize).toList

    val spanBatch = this.orphanSpans ++ newSpanBatch
    println(s"Total of spans to process (including orphans):
     ↪  ${spanBatch.length}")
    this.orphanSpans = List[ProtoSpan]()

    val processedSpans = process(spanBatch)

    this.orphanSpans = processedSpans
      .collect { case Left(protoSpans) => protoSpans }
      .toList
      .flatten
    println(s"Got ${this.orphanSpans.size} spans unprocessed in this
     ↪  batch")

    processedSpans.collect { case Right(a) => a }
  }
}

object JaegerAPIClient {
  def apply(jaegerQueryHostPort: String) = new
   ↪  JaegerAPIClient(jaegerQueryHostPort: String)
}
```

B.1.2 Mapping ProtoBuf Data to Standard Java/Scala API

This object `Converter` is the Scala transposition to my own needs of the `Converter` class defined in `io.jaegertracing.analytics.model`. Then we define an object named Converter that hold helper functions translating protobuf objects into Java objects. Target Java objects are compatible with JVM 8 (matching Scala version

of the notebook) and are native Java objects (instead of Scala). This will parse:

- Jaeger SpanIDs and TracesIDs as `String`
- Protobuf Timestamps as `LocalDateTime`
- Jaeger Tags (typed Key-Value tuples) into `Map[String, String]`
- Jaeger Span References into `Tuple3(String, String, String)` where:
  - the first item represents TraceID
  - the second SpanID
  - the third the type of relationship (either "CHILD_OF" or "FOLLOWS_FROM")

```
import com.google.protobuf.{
  Timestamp => ProtoTimestamp,
  Duration => ProtoDuration,
  ByteString => ProtoByteString
}
import io.jaegertracing.api_v2.Model.{
  KeyValue => JaegerKeyValue,
  ValueType,
  SpanRef,
  SpanRefType
}
import io.jaegertracing.analytics.model.{Converter =>
↪  JaegerBaseConverter}
import collection.JavaConverters._

import java.time.{LocalDateTime, ZoneOffset, Duration}
import java.util.{List => JList}

object Converter {
  def toStringId(pBs: ProtoByteString): String =
  ↪  JaegerBaseConverter.toStringId(pBs)

  def toLocalDateTime(pTs: ProtoTimestamp): LocalDateTime =
  ↪  LocalDateTime.ofEpochSecond(pTs.getSeconds, pTs.getNanos,
  ↪  ZoneOffset.UTC)

  def toDuration(pDur: ProtoDuration): Duration =
  ↪  Duration.ofSeconds(pDur.getSeconds, pDur.getNanos)

  // Rewriting this method as it is defined as Private in
  ↪  JaegerBaseConverter
  def toMap(kvs: JList[JaegerKeyValue]): Map[String, String] = {
```

```
    for { kv <- kvs.asScala } yield
      kv.getVType match {
        case ValueType.STRING  => (kv.getKey, kv.getVStr)
        case ValueType.BOOL    => (kv.getKey, kv.getVBool.toString)
        case ValueType.INT64   => (kv.getKey, kv.getVInt64.toString)
        case ValueType.FLOAT64 => (kv.getKey,
        ↪  kv.getVFloat64.toString)
        case ValueType.BINARY  => (kv.getKey,
        ↪  kv.getVBinary.toStringUtf8)
        case _                 => (kv.getKey, "unrecognized")
      }
  } toMap


  // Definition of Reference pointers used in Spans
  case class ReferenceId(traceId: String, spanId: String, name:
  ↪  String)

  object ReferenceId {
    def apply(traceId: String, spanId: String, name: String):
    ↪  ReferenceId =
      new ReferenceId(traceId, spanId, name)

    def of(tuple3: (String, String, String)): ReferenceId =
      this.apply(traceId = tuple3._1, spanId = tuple3._2, name =
      ↪  tuple3._3)

    def of(sr: SpanRef): ReferenceId = sr.getRefType match {
      case SpanRefType.CHILD_OF =>
        this.apply(toStringId(sr.getTraceId),
        ↪  toStringId(sr.getSpanId), "CHILD_OF")
      case SpanRefType.FOLLOWS_FROM =>
        this.apply(toStringId(sr.getTraceId),
        ↪  toStringId(sr.getSpanId), "FOLLOWS_FROM")
      case _ =>
        this.apply(toStringId(sr.getTraceId),
        ↪  toStringId(sr.getSpanId), "unrecognized")
    }
  }

  def toReferenceIds(srs: JList[SpanRef]): List[ReferenceId] = {
    for { sr <- srs.asScala } yield ReferenceId.of(sr)
  } toList
}
```

B.2 Definition of the Analytics Trace-Data-Model

We define a set of Scala classes modeling Jaeger Traces, they will be used as Label in our future property graph and are instantiated from protobuf data. In general terms the model cans be summed up by the following statement:

- `Operation`: This class models the concept of Operation that are shared among Spans in a Trace.
- `Resource`: This class represents instances executing of a program, thus costing resources. OpenTelemetry Resource semantic is defined at `https://github.com/open-telemetry/opentelemetry-specification/blob/v0.6.0/specification/resource/semantic_conventions/README.md`
- `Span`: In OpenTelemetry Spans are the representation of the latency measurement (at a given startTime there is an associated operation duration), they bring also other relevant numerical data, like, for example, the number of byte exchanged over the network. OpenTelemetry definition of Span can be found at: `https://github.com/open-telemetry/opentelemetry-specification/blob/v0.6.0/specification/trace/api.md#span`
- `Trace`: Trace is the aggregation of Spans characterizing the propagation of Remote Procedure Calls or other network calls in a distributed system, this spans build a Directed Acyclic Dependency Graph of Resources.

Each class defines a static `.of()` method used to wrap the constructor with Protobuf data pre-processing, thus allowing to create model instances directly from API data.

B.2.1 Operation Entities

```
import io.jaegertracing.api_v2.Model.{Span => ProtoSpan}

case class Operation(name: String, attributes: Map[String, String])

object Operation {
  def apply(name: String, attributes: Map[String, String]) = new
  ↪  Operation(name, attributes)

  def of(ps: ProtoSpan): Operation =
    this.apply(ps.getOperationName,
              Converter.toMap(ps.getTagsList)
                        .filterKeys { k => ! (k contains "span") })
}
```

B.2.2 Resources Entities

```
import io.jaegertracing.api_v2.Model.{Span => ProtoSpan}
```

```
case class Resource(name: String, attributes: Map[String, String])

object Resource {
  def apply(name: String, attributes: Map[String, String]): Resource
  ↪  = new Resource(name, attributes)

  def of(ps: ProtoSpan): Resource = {
    val res = ps.getProcess
    this.apply(res.getServiceName, Converter.toMap(res.getTagsList))
  }
}
```

B.2.3 Span Entities

```
import io.jaegertracing.api_v2.Model.{Span => ProtoSpan}

case class Span(
  spanId: String,
  traceId: String,
  startTime: LocalDateTime,
  duration: Duration,
  attributes: Map[String, String],
  references: List[Converter.ReferenceId],
  operation: Operation,
  resource: Resource
)

object Span {
  def apply(
    spanId: String,
    traceId: String,
    startTime: LocalDateTime,
    duration: Duration,
    attributes: Map[String, String],
    references: List[Converter.ReferenceId],
    operation: Operation,
    resource: Resource
  ): Span =
    new Span(spanId, traceId, startTime, duration, attributes,
    ↪  references, operation, resource)

  def of(ps: ProtoSpan): Span =
    this.apply(
      spanId = Converter.toStringId(ps.getSpanId),
      traceId = Converter.toStringId(ps.getTraceId),
      startTime = Converter.toLocalDateTime(ps.getStartTime),
```

```
      duration = Converter.toDuration(ps.getDuration),
      attributes = Converter.toMap(ps.getTagsList).filterKeys { _
       ↪  contains "span" },
      references = Converter.toReferenceIds(ps.getReferencesList),
      operation = Operation.of(ps),
      resource = Resource.of(ps)
    )
}
```

B.2.4 Trace Entities

```
import io.jaegertracing.api_v2.Model.{Span => ProtoSpan}

// Definition of Trace as an aggregation of Spans
case class Trace(traceId: String, spans: List[Span])

object Trace {
  def apply(traceId: String, spans: List[Span]): Trace =
    new Trace(traceId, spans)

  def of(traceId: String, protoSpans: List[ProtoSpan]):
   ↪  Either[List[ProtoSpan], Trace] = {

    val spans: List[Span] = for {
      ps <- protoSpans
    } yield Span.of(ps)

    val spanReferences = for {
      span <- spans
      ref  <- span.references
    } yield ref

    val isCompleteTrace = spanReferences forall {
      spanRef => spans exists {
        span => span.traceId == spanRef.traceId && span.spanId ==
         ↪  spanRef.spanId
      }
    }

    val rootSpans = spans filterNot {
      span => spanReferences.map(_.spanId).contains(span.spanId)
    }

    if (isCompleteTrace && rootSpans.length == 1) {
      Right(this.apply(traceId, spans))
    } else {
```

```
      Left(protoSpans)
    }
  }
}
```

B.3 Graph Encoding

B.3.1 Defining the Property Graph Model

In the following, we define a singleton object `TraceMetaModel` that own several properties referencing the model defined in the section 2.3 Extracting a Structural Model from Traces.

```
import gremlin.scala._

object TraceMetaModel {
  // Vertices Labels
  val SpanLabel      = "Span"
  val PodLabel       = "Pod"
  val NodeLabel      = "Node"
  val ZoneLabel      = "Zone"
  val RegionLabel    = "Region"
  val NamespaceLabel = "Namespace"
  val ClusterLabel   = "Cluster"

  // Edges Types
  val References  = "REFERENCE"
  val ExecutesOn  = "EXECUTES_ON"
  val IsContained = "IS_CONTAINED"

  // Span Vertices properties
  val traceId   = Key[String]("traceId")
  val spanId    = Key[String]("spanId")
  val startTime = Key[String]("startTime")
  val duration  = Key[Long]("duration")
  val kind      = Key[String]("kind")

  // Resource Vertices properties
  val name = Key[String]("name")
  val ip   = Key[String]("ip")
  val uid  = Key[String]("uid")

  // Edge properties
  val serviceName = Key[String]("serviceName")
  val networkTime = Key[Long]("networkTime")
}
```

B.3.2 Encoding Process

The following code presents a functionnal and parallelizable approache to the process of encoding a Graph with the custom model presented in this thesis. The graph are created with the Gremlin Scala library and fits multiple graph backends. Still, while the graph works on a Neo4j or a Gremlin Server centralized backend, its purpose is to be used on in-memory graphs: because an heavy flow of traces can apply too much pressure on the backend. Once processed this graph will be *merged* into a centralized knowledge graph.

```
import gremlin.scala._
import collection.JavaConverters._
import java.util.{ List => JList, ArrayList => JArrayList, Map =>
↪  JMap, Set => JSet }
import java.time.{ LocalDateTime, Duration }

def fromTrace(t: Trace)(implicit graph: ScalaGraph): ScalaGraph = {
  import TraceMetaModel._
  implicit val g = graph.traversal
  val supportsTransactions =
  ↪  g.graph.features.graph.supportsTransactions

  val spanVertices: Map[String, Vertex] = {
    for (s <- t.spans) yield {
      // Adds a "Span" vertex with startTime and Duration attributes
      val spanVertex = graph + (SpanLabel,
        traceId -> s.traceId,
        spanId -> s.spanId,
        startTime -> s.startTime.toString,
        duration -> s.duration.toNanos,
        kind -> s.attributes.getOrElse("span.kind", "")
      )

      // Consistant creation / reuse of "Resource" vertices based on
      ↪  both label and attributes
      val resourceVertices = for {
        rks <- ResourceKind.fromAttributes(s.resource.attributes)
      } yield rks match {
        case Pod(n, i, u) => mergeVertex(PodLabel, name -> n, ip ->
        ↪  i, uid -> u)
        case Node(n, i)   => mergeVertex(NodeLabel, name -> n, ip ->
        ↪  i)
        case Zone(n)      => mergeVertex(ZoneLabel, name -> n)
        case Region(n)    => mergeVertex(RegionLabel, name -> n)
        case Namespace(n) => mergeVertex(NamespaceLabel, name -> n)
        case Cluster(n)   => mergeVertex(ClusterLabel, name -> n)
      }
```

```
      if (supportsTransactions) { g.graph.tx.commit }

      // Link each identified resources to the span with the
      ↪  relationship "EXECUTES_ON"
      resourceVertices foreach {
        resourceVertex => spanVertex --- (ExecutesOn, serviceName ->
         ↪  s.resource.name) --> resourceVertex
      }
      if (supportsTransactions) { g.graph.tx.commit }

      // Index Span Vertices by "SpanId" for later use (i.e. the
      ↪  creation of the REFERENCE realtionships)
      s.spanId -> spanVertex
    }
  } toMap

  // Iterate once again over spans to recreate the DAG of spans
  for {
    s <- t.spans
    ref <- s.references if ref.traceId == s.traceId
  } spanVertices(s.spanId) --- (References, kind -> ref.name) -->
  ↪  spanVertices(ref.spanId)
  if (supportsTransactions) { g.graph.tx.commit }

  graph
}


/** Helper function that eventually creates a new Node in the graph
  * matching the label `label` and attributes `attrs` if it does
  * not already exists */
def mergeVertex(label: String, attrs: KeyValue[String]*)(implicit g:
↪  TraversalSource): Vertex = {
  // Forge the gremlin query by accumulating `.has("key", value)`
  // after the  `g.V.hasLabel()` for all items in `attrs`
  val gremlinSearchQuery = attrs.foldLeft(g.V().hasLabel(label)) {
    case (acc, kv) => acc.has[String](kv.key, kv.value)
  }

  // Consume the query and return the Node if it already exists
  // otherwise the node is created
  gremlinSearchQuery.headOption match {
    case Some(v) => v
    case None    => g.graph + (label, attrs: _*)
  }
}
```

B.3.3 Graph Rewriting

Definition of functions doing graph transformations on Gremlin In-Memory Graphs:

- The function `projectDependencyOn(labels: String*)` looks for REFERENCE relations in the graph and applies the previous pattern to create the PROJECTED_REF between Pods, Nodes, Zones (defined by the `labels` argument).
- The function `subgraphOf(label: String)(implicit graph: ScalaGraph)` creates en Edge-induced subgraph.

```
import gremlin.scala._
import collection.JavaConverters._

def projectDependencyOn(labels: String*)(implicit graph:
↪  ScalaGraph): List[Edge] = {
  import TraceMetaModel._

  val g = graph.traversal

  // Gremlin Query to find all REFERENCES relationships and get the
  ↪  Source and Dest vertices
  val clientServerSpanVertices =
    g.V().hasLabel(SpanLabel).as("srcSpan")
      .out(References)
      .hasLabel(SpanLabel).as("dstSpan")
      .select("srcSpan", "dstSpan")
      .toList
      .map(_.asScala)

  val dependentResourcesEdges = for {
    label <- labels
    v     <- clientServerSpanVertices
  } yield {
    val srcSpanV: Vertex = v("srcSpan").asInstanceOf[Vertex]
    val dstSpanV: Vertex = v("dstSpan").asInstanceOf[Vertex]
    val srcResOpt: Option[Vertex] =
    ↪  g.V(srcSpanV).out(ExecutesOn).hasLabel(label).headOption
    val dstResOpt: Option[Vertex] =
    ↪  g.V(dstSpanV).out(ExecutesOn).hasLabel(label).headOption
    val tId  = srcSpanV.property(traceId).value
    val netT = dstSpanV.property(duration).value -
    ↪  srcSpanV.property(duration).value

    (srcResOpt, dstResOpt) match {
      case (Some(srcRes), Some(dstRes)) if srcRes != dstRes =>
```

```
        Some(srcRes --- ("PROJECTED_DEP", traceId -> tId,
        ↪  networkTime -> netT) --> dstRes)
      case _ => None
    }
  }

  dependentResourcesEdges collect { case Some(v) => v } toList
}

def subgraphOf(label: String)(implicit graph: ScalaGraph):
↪  ScalaGraph = {
  import TraceMetaModel._

  val g = graph.traversal
  val subgraphStepLabel = StepLabel[Graph]("subGraph")

  g.V().hasLabel(label).outE("PROJECTED_DEP")
    .subgraph(subgraphStepLabel)
    .cap(subgraphStepLabel)
    .head
}
```

B.4 Calculation of the Flow hierarchy metric

After the previous graph rewriting operations, the initial property graph made of multiple labels for vertices and edges is turned in a graph where all vertices have the same label and so have all the edges. As a result the graph after rewriting operations, on which the flow hierarchy metric is calculated, can be summed up as $G = (V, E)$ where each vertices represents an instance of a resource (e.g. a Pod, a Node, a Zone, a Cluster) and each edge a network communication between these resources and is characterized by a Network Latency property.

The calculation of the flow requires to know whether each edge is in a cycle or not; to do so, the selected approach relies on the identification of strongly connected components, indeed an edge is in a cycle if and only if it is in a strongly connected component.

```
import gremlin.scala._
import collection.JavaConverters._
import java.util.Arrays

def flowHierarchy(graph: ScalaGraph, weightKey: Key[Long] =
↪  Key[Long]("Falls back to 1")): Double = {
  import TraceMetaModel._

  val scc = stronglyConnectedComponents(graph)
  val g = scc.traversal
```

```
  val sccIndexedEdges =
    g.E.project(_(By(__.inV().properties("component")))
                .and(By(__.outV.properties("component")))
                .and(By(__.coalesce(_.value(weightKey),
                ↪   _.constant(1L)))) // get value from the `weight`
                ↪   key or fall back to 1 to create the unweighted
                ↪   formula
               ).toList
               .map {
                 case (kvSrc, kvDst, edgeWeight) =>
                   (kvSrc.value(), kvDst.value(), edgeWeight)
               }

  val weightOutOfCycles =
    sccIndexedEdges.filter { case (cSrc, cDst, _) => cSrc != cDst }
                   .map { case (_, _, edgeWeight) => edgeWeight }
                   .fold(0L) {_ + _}
                   .toDouble

  val totalWeight = sccIndexedEdges
                      .map { case (_, _, edgeWeight) => edgeWeight }
                      .fold(0L) {_ + _}
                      .toDouble

  if (totalWeight != 0) {
    weightOutOfCycles / totalWeight
  } else {
    1.0
  }
}

/** Scala traverser does not support the Connected Components
↪   Algorithm, this function is based on the Java Implementation and
↪   requires a to use an OLAP to compute the Strongly Connected
↪   Components (SCC), the computation of the `scc` variable should
↪   be async */
def stronglyConnectedComponents(graph: ScalaGraph): ScalaGraph = {
  import
  ↪   org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.`__`
  import org.apache.tinkerpop.gremlin.process.traversal._
  import org.apache.tinkerpop.gremlin.tinkergraph.structure._
  import
  ↪   org.apache.tinkerpop.gremlin.process.computer.traversal.step.map.ConnectedComponen

  val gJava = graph.asJava.traversal.withComputer()
```

```
  val scc =
    gJava.V()
      .connectedComponent()
        .`with`(ConnectedComponent.propertyName, "component")
        .`with`(ConnectedComponent.edges, __.outE("PROJECTED_DEP"))
      .project("id", "component")
        .by(__.id())
        .by("component")
      .toList()

  // Ugly Java-Scala Wrapper, no other solution found
  val vertexIdComponentTuples: List[(Long, Long)] = for { vertexAttr
   ↪  <- scc.asScala.toList } yield
    Arrays.asList(vertexAttr.values().toArray: _*).asScala.toList
     ↪  match {
      case vertexId :: componentId :: Nil =>
       ↪  vertexId.asInstanceOf[Long] ->
       ↪  componentId.asInstanceOf[String].toLong
      case _ => ??? // Should never happen, if so raise an exception
    }

  // Fall back to Gremlin-Scala OLTP
  val g = graph.traversal
  val component = Key[Long]("component")

  for {
    (vid, cid) <- vertexIdComponentTuples
  } g.V(vid).property(component -> cid).iterate

  graph
}
```

B.5 Building and Running the Pipeline

Establishing connection with the two ends of the data transformation pipeline. This case uses a Jaeger gRPC client and not a Kafka client.

```
val jaegerClient   =
↪  JaegerAPIClient(sys.env.getOrElse("JAEGER_API_URI",
↪  "jaeger:16686"))
val neo4jConnector =
↪  Neo4jGraphConnector(sys.env.getOrElse("NEO4J_BOLT_URI",
↪  "bolt://neo4j/7687"), "neo4j", sys.env.get("NEO4J_PASSWORD"))

println("Connected to the local Jaeger instance and to the local
↪  Neo4j instance")
```

Creating two classes for wrapping results of flow hierarchy calculation.

```
sealed trait TraceComputation

case class TraceComputationUnweighted(
        traceId: String,
        timestamp: String,
        duration: Long,
        hPod: Double,
        hNode: Double,
        hZone: Double
    ) extends TraceComputation

case class TraceComputationWeighted(
        traceId: String,
        timestamp: String,
        duration: Long,
        hPod: Double,
        hNode: Double,
        hZone: Double
    ) extends TraceComputation
```

The following code manually consumes 5000 spans from Jaeger and rebuilds traces while ensuring traces are complete. These traces are stored in the variable `traces`.

```
import io.jaegertracing.api_v2.Model.{Span => ProtoSpan}

val traces = jaegerClient.batchAggregateAs[Trace](
                batchSize = 5000,
                serviceName = "frontend",
                operationName = Some("Recv./cart/checkout"),
                process = (protoSpans) => protoSpans groupBy {
                  span => Converter.toStringId(span.getTraceId)
                } map {
                  case (traceId, traceProtoSpans) =>
                  ↪  Trace.of(traceId, traceProtoSpans.toList)
                }
            ) toList

println(s"Got a total of ${traces.size} traces out of 5000 spans")
```

Finally the `trace` variable is processed, each trace is encoded into an in-memory hierarchical graph, each resource layers are extracted into their own graph and the flow hierarchy is computed for each of these layers. Results of the computation are sent into a list of `TraceComputationUnweighted` and `TraceComputationWeighted` which are stored in Neo4j for convenience in this notebook, although this backend is not suitable for this purpose.

```
println(s"Processing traces:")

val traceGraphs: Seq[(TraceComputation, TraceComputation)] = for { t
↪  <- traces } yield {
  val rootSpan = t.spans find { _.operation.name ==
  ↪  "Recv./cart/checkout" } get
  implicit val graph = TinkerGraph.open.asScala
  println(s"• Trace ${t.traceId} has ${t.spans.size} spans started
  ↪  at ${rootSpan.startTime.toString} and had a duration of
  ↪  ${rootSpan.duration.toNanos} ns")

  // 1.   Fill the graph with vertices representing the trace
  fromTrace(t)

  // 2.1. Do the graph rewriting for each abstraction levels in the
  ↪  hierachical location model
  projectDependencyOn("Pod", "Node", "Zone")
  val podGraph  = subgraphOf("Pod")
  val nodeGraph = subgraphOf("Node")
  val zoneGraph = subgraphOf("Zone")

  // 2.2 Calculate the Flow Hierarchy
  val computationUnweighted = TraceComputationUnweighted(t.traceId,
  ↪  rootSpan.startTime.toString, rootSpan.duration.toNanos,
    flowHierarchy(podGraph), flowHierarchy(nodeGraph),
    ↪  flowHierarchy(zoneGraph))
  println(s"  Unweighted Flow Hierarchies computed: h_Pod =
  ↪  ${computationUnweighted.hPod} | h_Node =
  ↪  ${computationUnweighted.hNode} | h_Zone =
  ↪  ${computationUnweighted.hZone}")

  val computationWeighted = TraceComputationWeighted(t.traceId,
  ↪  rootSpan.startTime.toString, rootSpan.duration.toNanos,
    flowHierarchy(podGraph, Key[Long]("networkTime")),
    ↪  flowHierarchy(nodeGraph, Key[Long]("networkTime")),
    ↪  flowHierarchy(zoneGraph, Key[Long]("networkTime")))
  println(s"  Weighted Flow Hierarchies computed:   h_Pod =
  ↪  ${computationWeighted.hPod} | h_Node =
  ↪  ${computationWeighted.hNode} | h_Zone =
  ↪  ${computationWeighted.hZone}")

  // 3. persist the rewritten graphs in Neo4J
  Try(neo4jConnector.addTrace(computationUnweighted)) match {
    case Failure(e) => println(s"  Did not manage to save this
    ↪  Trace, an exception occured : ${e.getMessage}")
    case Success(_) => {
```

```
      println(s"  Trace and computations saved in Neo4j")
      neo4jConnector.addResourceGraph(podGraph, t.traceId)
      neo4jConnector.addResourceGraph(nodeGraph, t.traceId)
      neo4jConnector.addResourceGraph(zoneGraph, t.traceId)
    }
  }
  Try(neo4jConnector.addTrace(computationWeighted))
  graph.close()

  (computationUnweighted, computationWeighted)
}
traceGraphs
```

Bibliography

[Akoglu 2015] Leman Akoglu, Hanghang Tong and Danai Koutra. Graph based anomaly detection and description: A survey. Number 3. 2015. (Cited in page 89.)

[Al-Mutawa 2014] Hussain A. Al-Mutawa, Jens Dietrich, Stephen Marsland and Catherine McCartin. *On the Shape of Circular Dependencies in Java Programs.* In 2014 23rd Aust. Softw. Eng. Conf., pages 48–57. IEEE, apr 2014. (Cited in page 62.)

[Alrifai 2009] Mohammad Alrifai and Thomas Risse. *Combining global optimization with local selection for efficient QoS-aware service composition.* Proc. 18th Int. Conf. World wide web - WWW '09, page 881, 2009. (Cited in page 19.)

[Alrifai 2010] Mohammad Alrifai, Dimitrios Skoutas and Thomas Risse. *Selecting skyline services for QoS-based web service composition.* Proc. 19th Int. Conf. World wide web - WWW '10, no. October 2007, page 11, 2010. (Cited in page 19.)

[Anand 2020] Vaastav Anand, Matheus Stolet, Thomas Davidson, Ivan Beschastnikh, Tamara Munzner and Jonathan Mace. *Aggregate-Driven Trace Visualizations for Performance Debugging.* CoRR, oct 2020. (Cited in page 27.)

[Anderson 1972] P. W. Anderson. *More Is Different: Broken symmetry and the nature of the hierarchical structure of science.* Science (80-.)., vol. 177, no. 4047, pages 393–396, aug 1972. (Cited in page 61.)

[Ardelean 2018] Dan Ardelean, Amer Diwan and Chandra Erdman. Performance analysis of cloud applications. USENIX Association, nov 2018. (Cited in pages 20, 26, 27, 29, 66, and 117.)

[Bistarelli 2018] Stefano Bistarelli, Lars Kotthoff, Francesco Santini and Carlo Taticchi. *Containerisation and dynamic frameworks in ICCMA'19.* CEUR Workshop Proc., vol. 2171, no. September, pages 4–9, 2018. (Cited in pages vii and 13.)

[Böttger 2018] Timm Böttger, Felix Cuadrado, Gareth Tyson, Ignacio Castro and Steve Uhlig. *Open connect everywhere: A glimpse at the internet ecosystem through the lens of the netflix CDN.* Comput. Commun. Rev., vol. 48, no. 1, pages 28–34, 2018. (Cited in page 87.)

[Boutin 2014] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu and Lidong Zhou. *Apollo: Scalable and coordinated scheduling for cloud-scale computing.* Proc. 11th USENIX Symp. Oper. Syst. Des. Implementation, OSDI 2014, pages 285–300, 2014. (Cited in pages 16 and 20.)

[Breivold 2008] Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land and Stig Larsson. *Using dependency model to support software architecture evolution*. In 2008 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng. - Work., pages 82–91. IEEE, sep 2008. (Cited in page 62.)

[Brin 1998] Sergey Brin and Lawrence Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. In COMPUTER NETWORKS AND ISDN SYSTEMS, pages 107–117, 1998. (Cited in pages 91 and 131.)

[Cantor 2020] Lyn Cantor and Cam Cullen. *Phenomena Report COVID-19 Spotlight*. Technical Report May, 2020. (Cited in pages 1 and 112.)

[Cassé 2021] Clément Cassé, Pascal Berthou, Philippe Owezarski and Sebastien Josset. *Using Distributed Tracing to Identify Inefficient Resources Composition in Cloud Applications*. In IEEE 10th Int. Conf. Cloud Netw., Virtual, 2021. (Cited in pages 5, 79, and 113.)

[Cassé 2022] Clément Cassé, Pascal Berthou and Philippe Owezarski. *A Tracing Based Model to Identify Bottlenecks in Physically Distributed Applications*. In Int. Conf. Inf. Netw. (ICOIN 2022), Jeju Island, Korea (South), 2022. (Cited in pages 5, 101, 113, and 136.)

[Chaczko 2011] Zenon Chaczko, Venkatesh Mahadevan, Shahrzad Aslanzadeh and Christopher Mcdermid. *Availability and Load Balancing in Cloud Computing*. In Int. Conf. Comput. Softw. Model., volume 14, pages 134–140, Singapore, 2011. IACSIT Press. (Cited in pages 15 and 115.)

[Chou 2019] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis and Lin Xiao. *Taiji : Managing Global User Traffic for Large-Scale Internet Services at the Edge*. SOSP '19 Proc. 27th ACM Symp. Oper. Syst. Princ., pages 430–446, 2019. (Cited in pages 15, 20, 26, and 115.)

[Chow 2014] Michael Chow, David Meisner, Jason Flinn, Daniel Peek and Thomas F. Wenisch. *The mystery machine: End-to-end performance analysis of large-scale Internet services*. Proc. 11th USENIX Symp. Oper. Syst. Des. Implementation, OSDI 2014, pages 217–231, 2014. (Cited in pages 25 and 116.)

[Clauset 2008] Aaron Clauset, Cristopher Moore and M. E. J. Newman. *Hierarchical structure and the prediction of missing links in networks*. Nature, vol. 453, no. 7191, pages 98–101, may 2008. (Cited in page 61.)

[Cockcroft 2016a] Adrian Cockcroft. *Evolution of Microservices - Craft Conference*, 2016. (Cited in pages viii, 21, 93, and 116.)

[Cockcroft 2016b] Adrian Cockcroft. *Microservices Workshop - Craft Conference*, 2016. (Cited in pages viii, 21, 93, and 116.)

[Da Cunha Rodrigues 2016] G. Da Cunha Rodrigues, R.N. Calheiros, V.T. Guimaraes, G.L. Dos Santos, M.B. De Carvalho, L.Z. Granville, L.M.R.

Tarouco and Rajkumar Buyya. *Monitoring of cloud computing environments: Concepts, solutions, trends, and future directions*. Proc. ACM Symp. Appl. Comput., vol. 04-08-Apri, pages 378–383, 2016. (Cited in page 28.)

[Dabbagh 2015a] Mehiar Dabbagh, Bechir Hamdaoui, Mohsen Guizani and Ammar Rayes. *Efficient datacenter resource utilization through cloud resource overcommitment*. Proc. - IEEE INFOCOM, vol. 2015-Augus, pages 330–335, 2015. (Cited in pages 16 and 115.)

[Dabbagh 2015b] Mehiar Dabbagh, Bechir Hamdaoui, Mohsen Guizani and Ammar Rayes. *Toward energy-efficient cloud computing: Prediction, consolidation, and overcommitment*. IEEE Netw., vol. 29, no. 2, pages 56–61, 2015. (Cited in pages 16 and 115.)

[Dalmazo 2017] Bruno L. Dalmazo, João P. Vilela and Marilia Curado. *Performance Analysis of Network Traffic Predictors in the Cloud*. J. Netw. Syst. Manag., vol. 25, no. 2, pages 290–320, apr 2017. (Cited in pages 25 and 116.)

[Dean 2014] Daniel J Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu and North Carolina. *PerfCompass: toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds*. Proc. 6th USENIX Conf. Hot Top. Cloud Comput., page 16, 2014. (Cited in pages 25 and 116.)

[Deng 2017] Jie Deng, Gareth Tyson, Felix Cuadrado and Steve Uhlig. *Internet Scale User-Generated Live Video Streaming : The Twitch Case*. In Passiv. Act. Meas., pages 60–71, 2017. (Cited in page 87.)

[Drewes 2000] Frank Drewes, Berthold Hoffmann and Detlef Plump. *Hierarchical graph transformation*. Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 1784 LNCS, pages 98–113, 2000. (Cited in page 50.)

[Durugbo 2013] Christopher Durugbo, Ashutosh Tiwari and Jeffrey R. Alcock. *Modelling information flow for organisations: A review of approaches and future challenges*. Int. J. Inf. Manage., vol. 33, no. 3, pages 597–610, 2013. (Cited in page 43.)

[El-Gazzar 2016] Rania El-Gazzar, Eli Hustad and Dag H. Olsen. *Understanding cloud computing adoption issues: A Delphi study approach*. J. Syst. Softw., vol. 118, pages 64–84, 2016. (Cited in pages 12 and 114.)

[Fowler 2014] Martin Fowler and James Lewis. *Microservices, a definition of this new architectural term*, 2014. (Cited in pages 18, 19, and 115.)

[Francis 2018] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer and Andrés Taylor. *Cypher: An evolving query language for property graphs*. In Proc. ACM SIGMOD Int. Conf. Manag. Data, pages 1433–1445, 2018. (Cited in pages 46 and 48.)

[Freeman 1979] L. C. Freeman. *Centrality in social networks.* Soc. Networks, vol. 1, no. 3, pages 215–239, 1979. (Cited in page 89.)

[Gan 2018a] Yu Gan, Meghna Pancholi, Dailun Cheng, Siyuan Hu, Yuan He and Christina Delimitrou. *Seer: Leveraging Big Data to Navigate the Increasing Complexity of Cloud Debugging.* In 10th USENIX Work. Hot Top. Cloud Comput. (HotCloud 18), apr 2018. (Cited in pages 25, 29, and 116.)

[Gan 2018b] Yu Gan, Meghna Pancholi, Dailun Cheng, Siyuan Hu, Yuan He and Christina Delimitrou. *Seer: Leveraging big data to navigate the increasing complexity of cloud debugging.* arXiv, 2018. (Cited in page 92.)

[Gan 2020] Yu Gan, Sundar Dev, David Lo and Christina Delimitrou. *Sage: Leveraging ML To Diagnose Unpredictable Performance in Cloud Microservices.* ML Comput. Archit. Syst., 2020. (Cited in pages 25 and 116.)

[Gluck 2020] Adam Gluck. *Introducing Domain-Oriented Microservice Architecture*, 2020. (Cited in pages 18, 19, 21, and 115.)

[Gonigberg 2018] Arthur Gonigberg, Mikey Cohen, Michael Smith, Gaya Varadarajan, Sudheer Vinukonda and Susheel Aroskar. *Open Sourcing Zuul 2*, 2018. (Cited in pages 83 and 87.)

[Grandi 2016] Robert Grandi, Mosharaf Chowdhury, Aditya Akella and Ganesh Ananthanarayanan. *Altruistic scheduling in multi-resource clusters.* Proc. 12th USENIX Symp. Oper. Syst. Des. Implementation, OSDI 2016, pages 65–80, 2016. (Cited in page 20.)

[Gud 2019] Amit Gud. *Testing in Production at Scale.* In SRECon19 _ Am. (USENIX Assoc., Brooklyn, NY, 2019. USENIX Association. (Cited in pages 21 and 23.)

[Haddad 2018] Einas Haddad. *Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow*, 2018. (Cited in page 21.)

[Heger 2017] Christoph Heger, André van Hoorn, Mario Mann and Dušan Okanović. *Application Performance Management.* Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. - ICPE '17, pages 429–432, 2017. (Cited in page 29.)

[Heinrich 2016] Robert Heinrich. *Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications Categories and Subject Descriptors.* Perform. Eval. Rev., vol. 43, no. 4, pages 13–22, 2016. (Cited in pages 8 and 16.)

[Heinrich 2017] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte and Johannes Wettinger. *Performance Engineering for Microservices.* Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. Companion - ICPE '17 Companion, pages 223–226, 2017. (Cited in pages 17, 19, 28, and 115.)

[Holten 2006] Danny Holten. *Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data*. IEEE Transactions on Visualization and Computer Graphics, vol. 12, no. 5, pages 741–748, 2006. (Cited in page 106.)

[Ibidunmoye 2015] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez and Erik Elmroth. *Performance Anomaly Detection and Bottleneck Identification*. ACM Comput. Surv., vol. 48, no. 1, pages 1–35, 2015. (Cited in pages 1, 29, 92, and 132.)

[IBM 2005] IBM. An architectural blueprint for autonomic computing. Number 3. IBM Whitepapers, 2005. (Cited in page 19.)

[Jayathilaka 2017] Hiranya Jayathilaka, Chandra Krintz and Rich Wolski. *Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications*. Proc. 26th Int. Conf. World Wide Web - WWW '17, pages 469–478, 2017. (Cited in pages 16, 26, 27, 30, 115, and 117.)

[Jeong 2001] H. Jeong, S. P. Mason, A.-L. Barabási and Z. N. Oltvai. *Lethality and centrality in protein networks*. Nature, vol. 411, no. 6833, pages 41–42, may 2001. (Cited in page 89.)

[Kaldor 2017] Jonathan Kaldor, Jonathan Mace, Micha Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan and Yee Jiun Song. *Canopy: An End-to-End Performance Tracing And Analysis System*. SOSP 2017 - Proc. 26th ACM Symp. Oper. Syst. Princ., pages 34–50, 2017. (Cited in page 20.)

[Kanzhelev 2020] Sergey Kanzhelev, Morgan McLean, Alois Reitbauer, Bogdan Drutu, Nik Molnar and Yuri Shkuro. *W3C Recommendation on Trace Context*, 2020. (Cited in pages 27 and 117.)

[Kendall 1994] Samuel C. Kendall, Jim Waldo, Ann Wollrath and Geoff Wyant. *A Note on Distributed Computing*. Technical Report, Sun Microsystems, Inc., 1994. (Cited in page 24.)

[Khan 2019] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob and Arif Ahmed. *Edge computing: A survey*. Futur. Gener. Comput. Syst., vol. 97, pages 219–235, 2019. (Cited in pages 15 and 115.)

[Klein 2010] Douglas Klein. *Centrality measure in graphs*. Journal of Mathematical Chemistry, vol. 47, pages 1209–1223, 05 2010. (Cited in page 89.)

[Kumar 2018] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim and Robert Soulé. *Semi-Oblivious Traffic Engineering : The Road Not Taken*. 15th USENIX Symp. Networked Syst. Des. Implement. (NSDI 18), pages 157–170, 2018. (Cited in pages 26, 27, and 116.)

[Las-Casas 2018] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes and Rodrigo Fonseca. *Weighted Sampling of Execution Traces.* In Proc. ACM Symp. Cloud Comput. - SoCC '18, pages 326–332, 2018. (Cited in pages 27 and 105.)

[Las-Casas 2019] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand and Jonathan Mace. *Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering.* SoCC 2019 - Proc. ACM Symp. Cloud Comput., pages 312–324, 2019. (Cited in page 27.)

[Lee 2003] Edward A Lee. *Model-driven development-from object-oriented design to actor-oriented design.* In Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (aka The Monterey Workshop), Chicago. Citeseer, 2003. (Cited in page 93.)

[Li 2019] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao and Yanbo Han. *Service Mesh: Challenges, state of the art, and future research opportunities.* Proc. - 13th IEEE Int. Conf. Serv. Syst. Eng. SOSE 2019, 10th Int. Work. Jt. Cloud Comput. JCC 2019 2019 IEEE Int. Work. Cloud Comput. Robot. Syst. CCRS 2019, pages 122–127, 2019. (Cited in pages 23 and 73.)

[Lin 2018] Jinjin Lin, Pengfei Chen and Zibin Zheng. *Microscope: Pinpoint performance issues with causal graphs in micro-service environments.* In Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), volume 11236 LNCS, pages 3–20, 2018. (Cited in pages 26, 30, and 117.)

[Luo 2011] Jianxi Luo and Christopher L. Magee. *Detecting evolving patterns of self-organizing networks by flow hierarchy measurement.* Complexity, vol. 16, no. 6, pages 53–61, jul 2011. (Cited in pages vii, 60, 61, 62, 64, 65, 66, 124, and 125.)

[Mace 2018a] Jonathan Mace and Rodrigo Fonseca. *Universal context propagation for distributed system instrumentation.* In Proc. Thirteen. EuroSys Conf., pages 1–18, New York, NY, USA, apr 2018. ACM. (Cited in page 27.)

[Mace 2018b] Jonathan Mace, Ryan Roelke and Rodrigo Fonseca. *Pivot Tracing.* ACM Trans. Comput. Syst., vol. 35, no. 4, pages 1–28, 2018. (Cited in page 27.)

[Malawski 2018] Maciej Malawski, Kamil Figiela, Adam Gajek and Adam Zima. *Benchmarking heterogeneous cloud functions.* Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 10659 LNCS, no. August, pages 415–426, 2018. (Cited in page 14.)

[Marvasti 2013] Mazda Marvasti, Arnak Poghosyan, Ashot Harutyunyan and Naira Grigoryan. *Identifying Root Causes, Bottlenecks, and Black Swans in IT Environments.* VMWARE Tech. J., vol. 2, no. 1, pages 35–45, 2013. (Cited in pages 29, 92, and 132.)

[Maurya 2019] Sunil Kumar Maurya, Xin Liu and Tsuyoshi Murata. *Fast Approximations of Betweenness Centrality with Graph Neural Networks*. In Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM '19, page 2149–2152, New York, NY, USA, 2019. Association for Computing Machinery. (Cited in page 91.)

[Mell 2011] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology*. Natl. Inst. Stand. Technol. Inf. Technol. Lab., vol. 145, page 7, 2011. (Cited in pages 9, 10, 12, and 114.)

[Moreno-Vozmediano 2017] R. Moreno-Vozmediano, R. S. Montero, E. Huedo and I. M. Llorente. *Orchestrating the deployment of high availability services on multi-zone and multi-cloud scenarios*. J. Grid Comput., vol. 16, no. 1, pages 39–53, 2017. (Cited in page 15.)

[Nedelkoski 2019] Sasho Nedelkoski, Jorge Cardoso and Odej Kao. *Anomaly detection and classification using distributed tracing and deep learning*. Proc. - 19th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. CCGrid 2019, pages 241–250, 2019. (Cited in pages 25 and 27.)

[Netflix 2014] Netflix. *Introducing Atlas: Netflix's Primary Telemetry Platform*, 2014. (Cited in page 21.)

[Newman 2015] Sam Newman. Building Microservices. O'Reilly Media, Inc., 2015. (Cited in pages 18, 19, and 115.)

[Parisi-Presicce 1993] Francesco Parisi-Presicce. *Single vs. Double pushout derivations of graphs*. Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 657 LNCS, pages 248–262, 1993. (Cited in page 45.)

[Quantcast 2013] Quantcast. *Wait, How Many Metrics ? Monitoring at Quantcast*, 2013. (Cited in pages 21, 25, and 116.)

[Rodriguez 2015] Marko A. Rodriguez. *The gremlin graph traversal machine and language (Invited Talk)*. DBPL 2015 - Proc. 15th Symp. Database Program. Lang., pages 1–10, 2015. (Cited in pages 45, 46, and 52.)

[Sales-Pardo 2007] M. Sales-Pardo, R. Guimera, A. A. Moreira and L. A. N. Amaral. *Extracting the hierarchical organization of complex systems*. Proc. Natl. Acad. Sci., vol. 104, no. 39, pages 15224–15229, sep 2007. (Cited in page 61.)

[Sampaio 2019] Adalberto R. Sampaio, Julia Rubin, Ivan Beschastnikh and Nelson S. Rosa. *Improving microservice-based applications with runtime placement adaptation*. J. Internet Serv. Appl., vol. 10, no. 1, 2019. (Cited in page 19.)

[Saxena 2017] Akrati Saxena, Ralucca Gera and S. R. S. Iyengar. *A Faster Method to Estimate Closeness Centrality Ranking*. CoRR, vol. abs/1706.02083, 2017. (Cited in page 91.)

[Senyo 2018] Prince Kwame Senyo, Erasmus Addae and Richard Boateng. *Cloud computing research: A review of research themes, frameworks, methods and future research directions.* Int. J. Inf. Manage., vol. 38, no. 1, pages 128–139, 2018. (Cited in pages 12 and 114.)

[Singh 2017] Sukhpal Singh, Inderveer Chana and Maninder Singh. *The Journey of QoS-Aware Autonomic Cloud Computing.* IT Prof., vol. 19, no. 2, pages 42–49, 2017. (Cited in pages 17 and 115.)

[Souders 2009] Steve Souders. *Velocity and the Bottom Line*, 2009. (Cited in page 1.)

[Stergiou 2020] Stergios Stergiou. *Scaling PageRank to 100 Billion Pages.* In WWW '20: The Web Conference, April 20–24, 2020, Taipei, Taiwan, 2020. (Cited in page 92.)

[Tarjan 1972] Robert Tarjan. *Depth-First Search and Linear Graph Algorithms.* SIAM J. Comput., vol. 1, no. 2, pages 146–160, jun 1972. (Cited in page 65.)

[Thalheim 2017] Jorg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao and Christof Fetzer. *Sieve: Actionable insights from monitored metrics in distributed systems.* Middlew. 2017 - Proc. 2017 Int. Middlew. Conf., pages 14–27, 2017. (Cited in pages 25 and 116.)

[Uber 2016] Uber. *Observability at Uber Engineering: Past, Present, Future*, 2016. (Cited in pages 21 and 115.)

[Unuvar 2015] Merve Unuvar, Stefania Tosi, Yurdaer N. Doganata, Malgorzata Steinder and Asser N. Tantawi. *Selecting Optimum Cloud Availability Zones by Learning User Satisfaction Levels.* IEEE Trans. Serv. Comput., vol. 8, no. 2, pages 199–211, 2015. (Cited in pages 15 and 115.)

[van Eyk 2018] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann and Simon Eismann. *A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures.* Companion 2018 ACM/SPEC Int. Conf. Perform. Eng. - ICPE '18, pages 21–24, 2018. (Cited in pages 14 and 115.)

[Vaquero 2008] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres and Maik Lindner. *A break in the clouds.* ACM SIGCOMM Comput. Commun. Rev., vol. 39, no. 1, page 50, 2008. (Cited in pages 9 and 114.)

[Varghese 2018] Blesson Varghese and Rajkumar Buyya. *Next generation cloud computing: New trends and research directions.* Futur. Gener. Comput. Syst., vol. 79, pages 849–861, 2018. (Cited in pages 14, 15, and 115.)

[Vavilapalli 2013] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason

Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed and Eric Baldeschwieler. *Apache hadoop YARN: Yet another resource negotiator*. Proc. 4th Annu. Symp. Cloud Comput. SoCC 2013, 2013. (Cited in page 16.)

[Veeraraghavan 2016] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain and Dmitri Perelman. *Kraken : Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services*. OSDI'16 Proc. 12th USENIX Conf. Oper. Syst. Des. Implement., pages 635–651, 2016. (Cited in pages 20, 26, 92, 116, and 132.)

[Veeraraghavan 2018] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song and Tianyin Xu. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. 2018. (Cited in pages 20, 26, 92, 116, and 132.)

[Verma 2015] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune and John Wilkes. *Large-scale cluster management at Google with Borg*. In Proc. 10th Eur. Conf. Comput. Syst. EuroSys 2015, pages 1–17, New York, New York, USA, 2015. ACM Press. (Cited in pages 16 and 22.)

[Woodruff 2017] David P Woodruff and Qin Zhang. *When distributed computation is communication expensive*. Distrib. Comput., vol. 30, no. 5, pages 309–323, 2017. (Cited in page 24.)

[Yu 2018] Zhibin Yu and Qixiao Liu. *The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload : a View from Alibaba Trace*. In Proc. ACM Symp. Cloud Comput. - SoCC '18, pages 347 – 360, 2018. (Cited in pages 16, 20, and 115.)

[Zafeiris 2018] Anna Zafeiris and Tamás Vicsek. Why We Live in Hierarchies? Number July in SpringerBriefs in Complexity. Springer International Publishing, Cham, 2018. (Cited in pages 41, 42, 61, 108, and 123.)

Abstract:

Cloud Computing has changed how software is developed and deployed. Nowadays, Cloud applications are designed as rapidly evolving distributed systems that are hosted in third-party data centre and potentially scattered around the globe. This shift of paradigms also had a considerable impact on how software is monitored: Cloud application have been growing to reach the scale of hundreds of services, and state-of-the-art monitoring quickly faced scaling issues. In addition, monitoring tools also now have to address distributed systems failures, like partial failures, configuration inconsistencies, networking bottlenecks or even noisy neighbours.

In this thesis we present an approach based on a new source of telemetry that has been growing in the realm of Cloud application monitoring. Indeed, by leveraging the recent OpenTelemetry standard, we present a system that converts "distributed tracing" data in a hierarchical property graph. With such a model, it becomes possible to highlight the actual topology of Cloud applications like the physical distribution of its workloads in multiple data centres. The goal of this model is to exhibit the behaviour of Cloud Providers to the developers maintaining and optimizing their application.

Then, we present how this model can be used to solve some prominent distributed systems challenges: the detection of inefficient communications and the anticipation of hot points in a network of services. We tackle both of these problems with a graph-theory approach. Inefficient composition of services is detected with the computation of the Flow Hierarchy index. A Proof of Concept is presented based on a real OpenTelemetry instrumentation of a Zonal Kubernetes Cluster. In, a last part we address the concern of hot point detection in a network of services through the perspective of graph centrality analysis. This work is supported by a simulation program that has been instrumented with OpenTelemetry in order to emit tracing data. These traces have been converted in a hierarchical property graph and a study on the centrality algorithms allowed to identify choke points.

Both of the approaches presented in this thesis comply with state-of-the-art Cloud application monitoring. They propose a new usage of Distributed Tracing not only for investigation and debugging but for automatic detection and reaction on a full system.

Keywords: Cloud Application, Performance Monitoring, Distributed Tracing, Flow Hierarchy, Centrality Analysis, Property Graphs

Résumé :

Le Cloud Computing a bouleversé la façon dont nous développons et déployons les logiciels. De nos jours, les applications Cloud sont conçues comme des systèmes distribués en permanente évolution, hébergés dans des data center, et potentiellement même dispersés dans le monde entier. Ce changement de paradigme a également eu un impact considérable sur la façon dont les logiciels sont monitorés : les applications Cloud peuvent se composer de plusieurs centaines de services, et les outils de monitoring ont rapidement rencontré des problèmes de passage à l'échelle. De plus, ces outils de monitoring doivent désormais également traiter les défaillances et les pannes inhérentes aux systèmes distribués, comme par exemple, les pannes partielles, les configurations incohérentes, les goulots d'étranglement ou même la vampirisation de ressources.

Dans cette thèse, nous présentons une approche basée sur une nouvelle source de télémétrie qui s'est développée dans le domaine du monitoring des applications Cloud. En effet, en nous appuyant sur le récent standard OpenTelemetry, nous présentons un système qui convertit les données de “traces distribuées” en un graphe de propriétés hiérarchique. Grâce à ce modèle, il devient possible de mettre en évidence la topologie des applications, y compris sur plusieurs data-centers. L'objectif de ce modèle est donc d'exposer le comportement des fournisseurs de service Cloud aux développeurs qui maintiennent et optimisent leur application.

Ensuite, nous présentons l'utilisation de ce modèle pour résoudre certains des défis majeurs des systèmes distribués : la détection des communications inefficaces entre les services ainsi que l'anticipation des goulots d'étranglement. Nous abordons ces deux problèmes avec une approche basée sur la théorie des graphes. La composition inefficace des services est détectée avec le calcul de l'indice de hiérarchie de flux. Une plateforme Proof-of-Concept représentant un cluster Kubernetes zonal pourvu d'une instrumentation OpenTelemetry est utilisée pour créer et détecter les compositions de services inefficaces. Dans une dernière partie, nous abordons la problématique de la détection des goulots d'étranglement dans un réseau de services au travers de l'analyse de centralité du graphe hiérarchique précédent. Ce travail s'appuie sur un programme de simulation qui a aussi été instrumenté avec OpenTelemetry afin d'émettre des données de traçage. Ces traces ont été converties en un graphe de propriétés hiérarchique et une étude sur les algorithmes de centralité a permis d'identifier les points d'étranglement.

Les deux approches présentées dans cette thèse utilisent et exploitent l'état de l'art en matière de monitoring des applications Cloud. Elles proposent une nouvelle utilisation des données de “distributed tracing” pas uniquement pour l'investigation et le débogage, mais pour la détection et la réaction automatiques sur un système réel.

Mots Clés : Applications Cloud, Analyse de Performances, Traces Distribuées, Hiérarchie de Flux, Analyse de Centralité, Graphes de Propriétés