On the identification of performance bottlenecks in multi-tier distributed systems

Maha Alsayasneh

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Maha ALSAYASNEH

Thèse dirigée par **Noël De Palma, Professeur, Université Grenoble Alpes**

préparée au sein du **Laboratoire d'Informatique de Grenoble** dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

On the Identification of Performance Bottlenecks in Multi-tier Distributed Systems

Thèse soutenue publiquement le **« 15 mai 2020 »**,
devant le jury composé de :

Madame Sihem AMER-YAHIA
Directrice de recherche, CNRS, Université Grenoble Alpes, Présidente
Monsieur Noel DE PALMA
Professeur, Université Grenoble Alpes, Directeur de thèse
Monsieur Daniel HAGIMONT
Professeur, INPT/ENSEEIHT, Rapporteur
Monsieur Pierre SENS
Professeur, Sorbonne Université, Rapporteur

This thesis is dedicated to the memory of my father, Faisel Alsayasneh.
I miss him every day, but I am sure that he is glad and proud of me and my success.
He always believed in me.

Maha Alsayasneh
Grenoble, June 10, 2020

Acknowledgements

First of all, I would like to express my sincere gratitude to the members of my Ph.D. thesis jury, Dir. Sihem Amer-Yahia, Pr. Daniel Hagimont, and Pr. Pierre Sens, which have taken the time to examine my work and made meaningful comments and remarks before and during the defense. It was an honor for me to present my work in front of such a prestigious jury. I would also like to express my sincere appreciation to my Ph.D. advisor, Pr. Noel De Palma, who convincingly guided and encouraged me to be professional and do the right thing even when the road got tough. Without his persistent help, the goal of this Ph.D. would not have been realized.

This thesis would not have happened without the financial support of the EU AURA FEDER Project Studio Virtuel.

I would like to thank the members of the ERODS team for accepting nothing less than excellence from me. Without their support, this Ph.D. would have been less pleasurable. Thus, I want to thank (in alphabetical order): Diego Perez Arroyo, Thomas Calmant, Matthieu Caneill, Amadou Diarra, Ahmed El Rheddhane, Christopher Ferreira, Soulaimane Guedria, Hugo Guiroux, Vikas Jaiman, Ana Khorguani, Vincent Lamotte, Thomas Lavocat, Vania Marangozova-Martin, Muriel Nguimtsa, Albin Petit, Marc Platini, Felix Renard and Thomas Ropars.

I would also like to extend my gratitude to Vincent Leroy. Without your support on both personal and professional levels, I would never be here.

Special thanks to my wonderful friends that I have shared happy, crazy, and pleasant moments with, especially Jad, Sami, and Sara. You always believed in me, even when I was not. Thank you for everything that you have done for me.

Thanks to my many friends who become a family to me, you should know that your faith in me worths more than I can express on paper. Thanks to Wajdi, Cosette, Sofi, Eyad, and Omar.

Nobody has been more important to me in the pursuit of this thesis than my family. I would like to thank my mom, whose love and guidance are with me in whatever I pursue. She is the ultimate role model. Thanks to my brothers for supporting me spiritually throughout writing this thesis and my life in general. Most importantly, I would like to express my deepest appreciation to my loving and supportive boyfriend, Thomas, who always provides me unending inspiration. My success would not have been possible without your support and encouragement.

Maha Alsayasneh
Grenoble, June 10, 2020

Preface

This thesis presents the research conducted in the ERODS team at Laboratoire d'Informatique de Grenoble, to pursue a Ph.D. in Computer Science from the doctoral school "Mathématiques, Sciences et Technologies de l'Information, Informatique" of the Université Grenoble-Alpes. My research activities have been supervised by Noel De Palma (ERODS / Université Grenoble Alpes). This thesis investigates whether it is possible to identify a set of key metrics that can be used as reliable and general indicators of performance bottlenecks. It also shows how to automatically and accurately determine if the multi-tier distributed system has reached its maximum capacity in terms of throughput.

This work has led to the following publication:

- Maha Alsayasneh and Noel De Palma. "A Learning-Based Approach for Evaluating the Capacity of Data Processing Pipelines". European Conference on Parallel Processing 2020 (to appear).

Grenoble, Spring 2020.

Abstract

Today's distributed systems are made of various software components with complex interactions and a large number of configuration settings. Pinpointing the performance bottlenecks is generally a non-trivial task, which requires human expertise as well as trial and error. Moreover, the same software stack may exhibit very different bottlenecks depending on factors such as the underlying hardware, the application logic, the configuration settings, and the operating conditions. This work aims to (i) investigate whether it is possible to identify a set of key metrics that can be used as reliable and general indicators of performance bottlenecks, (ii) identify the characteristics of these indicators, and (iii) build a tool that can automatically and accurately determine if the system reaches its maximum capacity in terms of throughput.

In this thesis, we present three contributions. First, we present an analytical study of a large number of realistic configuration setups of multi-tier distributed applications, more specifically focusing on data processing pipelines. By analyzing a large number of metrics at the hardware and at the software level, we identify the ones that exhibit changes in their behavior at the point where the system reaches its maximum capacity. We consider these metrics as reliable indicators of performance bottlenecks. Second, we leverage machine learning techniques to build a tool that can automatically identify performance bottlenecks in the data processing pipeline. We consider different machine learning methods, different selections of metrics, and different cases of generalization to new setups. Third, to assess the validity of the results obtained considering the data processing pipeline for both the analytical and the learning-based approaches, the two approaches are applied to the case of a Web stack.

From our research, we draw several conclusions. First, it is possible to identify key metrics that act as reliable indicators of performance bottlenecks for a multi-tier distributed system. More precisely, identifying when the server has reached its maximum capacity can be identified based on these reliable metrics. Contrary to the approach adopted by many existing works, our results show that a combination of metrics of different types is required to ensure reliable identification of performance bottlenecks in a large number of setups. We also show that approaches based on machine learning techniques to analyze metrics can identify performance bottlenecks in a multi-tier distributed system. The comparison of different models shows that the ones based on the reliable metrics identified by our analytical study are the ones that achieve the best accuracy. Furthermore, our extensive analysis shows the robustness of the obtained models that can generalize to new setups, to new numbers of clients, and to both new setups and new numbers of clients. Extending the analysis to a Web stack confirms the main findings obtained through the study of the data processing pipeline. These results pave the way towards a general and accurate tool to identify performance bottlenecks in distributed systems.

Résumé

De nos jours, les systèmes distribués sont constitués de nombreux composants logiciels ayant des interactions complexes et de nombreuses possibilités de configurations. Dès lors, localiser les problèmes de performance est une tâche difficile, nécessitant une expertise humaine et de nombreux essais. En effet, la même pile logicielle peut se comporter différemment en fonction du matériel utilisé, de la logique applicative, des paramètres de configuration et des conditions de fonctionnement. Ce travail a pour objectif (i) d'identifier un ensemble de métriques de référence, générales et fiables, pour localiser les problèmes de performance, (ii) d'identifier les caractéristiques de ces indicateurs, et (iii) de construire un outil qui puisse déterminer de manière automatique si le système a atteint sa capacité maximale en terme de débit.

Dans cette thèse, nous présentons trois contributions principales. Premièrement, nous présentons une étude analytique d'un grand nombre de configurations réalistes d'applications distribuées multi-tiers, se concentrant sur les chaînes de traitements des données. En analysant un grand nombre de métriques au niveau logiciel et matériel, nous identifions celles dont le comportement change au moment où le système atteint sa capacité maximale. Deuxièmement, nous exploitons les techniques d'apprentissage machine pour développer un outil capable d'identifier automatiquement les problèmes de performance dans la chaîne de traitement de données. Pour ce faire, nous évaluons plusieurs techniques d'apprentissage machine, plusieurs sélections de métriques, et différents cas de généralisation pour de nouvelles configurations. Troisièmement, pour valider les résultats obtenus sur la chaîne de traitement de données, nous appliquons notre approche analytique et notre approche fondée sur l'apprentissage machine au cas d'une architecture Web.

Nous tirons plusieurs conclusions de nos travaux. Premièrement, il est possible d'identifier des métriques clés qui sont des indicateurs fiables de problèmes de performance dans les systèmes distribués multi-tiers. Plus précisément, ces métriques fiables permettent d'identifier le moment où le serveur a atteint sa capacité maximale. Contrairement à l'approche adoptée par de nombreux travaux existants, nos travaux démontrent qu'une combinaison de métriques de différents types est nécessaire pour assurer une identification fiable des problèmes de performance dans un grand nombre de configurations. Nos travaux montrent aussi que les approches fondées sur des méthodes d'apprentissage machine pour analyser les métriques permettent d'identifier les problèmes de performance dans les systèmes distribués multi-tiers. La comparaison de différents modèles met en évidence que ceux utilisant les métriques fiables identifiées par notre étude analytique sont ceux qui obtiennent la meilleure précision. De plus, notre analyse approfondie montre la robustesse des modèles obtenues. En effet, ils permettent une généralisation à de nouvelles configurations, à de nouveaux nombres de clients, et à de nouvelles configurations exécutées avec de nouveaux nombres de clients. L'extension de notre étude au cas d'une architecture Web confirme les résultats principaux obtenus à travers l'étude sur la chaîne de traitement de données. Ces résultats ouvrent la voie à la construction d'un outil générique pour identifier de manière fiable les problèmes de performance dans les systèmes distribués.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Context

Applications deployed in distributed environments are composed of a variety of software components. These components provide different functionalities e.g., publish-subscribe messaging, real-time analysis and rendering of streaming data, and storage.

To achieve scalability, each component can be divided into a number of partitions spread on separate machines for parallel processing. Additionally, for fault tolerance and high availability, each component or partition of a component typically has a number of replicas. These components (and their internal replicas and partitions) have many interactions, involving both control messages and data. With such a complex and diverse architecture, it is generally difficult to understand the overall behavior of a distributed system and how its performance can be improved. Moreover, pinpointing the performance bottlenecks in that context is a non-trivial task, which requires human expertise as well as trial and error.

The first step of a performance optimization or troubleshooting process for a deployed system usually consists of pinpointing in which part of the distributed system lies the main bottleneck. In many cases, solving a simple version of this problem is clearly challenging: by observing a pair of interacting component types (for example, a set of clients issuing requests to a set of replicated message brokers), how to determine which side is currently limiting the performance of the system? Answering this question is generally not straightforward, especially if one wants to avoid resorting to a *brute-force* approach via a campaign of performance characterization experiments. Such campaigns are typically (i) time-consuming, (ii) difficult to run faithfully on in-vitro testbeds, and (iii) intrusive to run on production systems.

An ideal alternative would be to identify a small set of indicators allowing to quickly and reliably hint at the limiting component. However, the current situation is very far from this ideal. While there exists a large number of methodologies and tools for pinpointing performance problems in distributed systems, there

is no consensus on which technique(s) to use in a given situation. Different bottlenecks typically exhibit different symptoms (e.g., resource saturation versus idle time). Moreover, the number of potential metrics to be considered (at the hardware, operating system and, application-level) is often very large. Performance engineers are often overwhelmed with data, yet left without clear and actionable insight.

Multi-tier architectures are commonly adopted in modern distributed systems. In a multi-tier architecture, the software components are organized in layers. A component in the level N can only interact with the components at the levels $N-1$ and $N+1$. Data processing pipelines and Web stacks are ubiquitous examples of multi-tier architectures. In this thesis, we study a data processing pipeline comprising Kafka, a distributed publish-subscribe messaging system, Spark Streaming, a Big Data processing engine and analysis framework, and Cassandra, a distributed NoSQL database management system. This software stack is widely used in production for analyzing streams of data [4, 9, 46, 49]. The second use case is a Web stack (LAMP) comprising two main components, the Apache server, a commonly used web server software, and MySQL, a relational database management system. The LAMP stack is broadly used in production to create web applications and dynamic websites [2, 4, 14, 64]. By considering the Web stack case, we aim to investigate if it is possible to generalize the findings obtained through the analysis of the data processing pipeline to another multi-tier architecture with different characteristics.

In this thesis, we consider the following problem: *First, is it possible to identify a set of key metrics that can be used as reliable indicators of performance bottlenecks on an already deployed multi-tier distributed system? By* reliable*, we mean that the indicator should provide accurate results despite significant changes in configuration setups. Second, based on the set of key metrics identified from the first question, is it possible to design a tool that is able to determine in a given setup configuration whether the server side of the application has reached its maximum capacity?*

The crux of the problem is that each studied case is different. Many factors can influence the behavior of a deployed multi-tier system. Among those factors, the architectural design and communication patterns of the components composing the multi-tier system can be of major importance. In the studied data processing pipeline, some components rely on a master-slave architecture while others adopt a peer-to-peer model, some communications are synchronous while others are asynchronous. Underlying hardware characteristics such as the number of cores per processor, or the network bandwidth, also have an impact on the performance. The configuration of the software components composing the multi-tier system is also to be taken into account. For instance, a change in the replication factor used by a component may modify its throughput. Last but not least, the performance of a multi-tier system depends on the complexity of the application logic and the characteristics of the data that are processed.

All these factors that can influence the behavior of distributed applications make it difficult to spot performance issues and to identify the causes of these issues. In this thesis, we are interested in the *global throughput* of the systems as the main performance metric. By *global throughput*, we refer to the amount of work (or tasks) that the system can handle per time unit. We aim at identifying when the system reaches its maximum throughput. The key challenge here is that the maximum throughput of the system is not known in advance because of all the factors described above. In other words, we are not interested in satisfying some objective functions, i.e., some service level objectives such as keeping the request-response time below a maximum bound [109, 121, 162]. Instead, we aim at improving the *global throughput* of the system without predefining the throughput value that we should achieve.

1.2 Objectives

Debugging and monitoring distributed systems is challenging. Understanding a system behavior requires gaining insight into the system. Several approaches have been designed to help to debug distributed systems [74]. System monitoring can be performed *off-line* where data about the system is collected and the analysis is done *off-line* (e.g., testing, model checking, and log analysis) [154], or *on-line* where a system specification is checked against an observed execution dynamically (e.g., tracing where we track the flow of data through a system, even across applications and protocols such as a database or a Web server [120]). However, each of the mentioned approaches has some limitations. For example, in addition to the fact that most testing of distributed systems is done using manually written tests, it is also limited to a certain number of executions and thus it can never guarantee to reveal all performance issues [74]. Even for the most recent monitoring approaches (e.g., tracing), which perform efficiently, it requires hard effort by the developers or the system users in instrumenting the systems to properly forward the tracing data. On the other hand, analyzing and modeling a system behavior is hard. Analyzing the collected data from the monitoring phase to find the performance limitations is challenging. Significant opportunities remain to be discovered [137]. Some research efforts focus on analyzing distributed systems logs [74]. It is a common black-box approach in which system console logs, debug logs, and other log sources are used to understand the system. However, while detailed logs from realistic systems contain valuable details, they tend to be so large that they are overwhelming to programmers, who as a result can not directly benefit from them. Other research works focus on analyzing a single type of metrics where a pre-defined objective function should be satisfied (i.e., response time metric should stay stable with a given value) [121]. Several works rely on examining the saturation of the system hardware resources (i.e., CPU, memory, network, and disk IO) [102, 117, 126]. This approach is probably the most used as it is fairly easy to use in production systems.

We posit that to simplify the work of the programmers and users of multi-tier architectures, it would be ideal to identify a small set of key metrics that can provide valuable information about the performance currently achieved by the system. Building a tool that can automatically identify performance issues based on the analysis of these metrics would be very helpful. Hence, our work aims at answering the following questions:

- Are there metrics that can be simply exported in a non-intrusive manner from a multi-tier distributed application, that can be used to identify performance bottlenecks in the system?
- In case of the existence of such reliable indicators of performance bottlenecks, what are these metrics? How can we identify them? What are their characteristics, i.e., how do these metrics behave? Do they belong to a specific type of metric (e.g, resource consumption metrics)?
- Are we able to rely on these reliable indicators to build a tool that can automatically determine if the server side of a multi-tier system has reached its maximum capacity in terms of global throughput? Can the predictions made by this tool be robust to significant changes in the setup?

1.3 Contributions

In this thesis, we have performed three contributions.

An analytical study to identify reliable indicators of performance bottlenecks

The first contribution (Chapter 3) is a broad study aiming at identifying reliable indicators of performance bottlenecks in the data processing pipeline. We propose an analysis to understand the relation between the system throughput and its metrics on both hardware and software levels.

We consider several configuration setups (e.g., different message sizes, different numbers of partitions, etc.), different workloads (e.g., a simple Wordcount application and a machine learning workload), as well as different deployments (e.g., with and without Cassandra). We study large scale deployments of this stack with up to 60 machines. We perform our experiments on top of Grid'5000 [1]. We export different kinds of metrics on both hardware and software levels for each component in the running system. We examine 53 metrics over more than 35 different system configurations. We study the evolution of these metrics with respect to the global throughput of the system. Our results show that (i) there are metrics that are *reliable* indicators of performance bottleneck in the data processing pipeline, (ii) a combination of a few metrics of different types (e.g., idle threads metrics and queue waiting time metrics) is sufficient to identify the limiting component in the system, and (iii) approaches relying on one single type of metrics (e.g., resource consumption or response time metrics) are not sufficient to identify the limiting component in the data processing pipeline.

A learning-based approach for performance bottleneck identification

The second contribution (Chapter 4) is a learning-based tool that applies machine learning classification algorithms to the key metrics identified by the previous contribution to automatically identify the limiting component in the data processing pipeline. We examine different learning models trained using different selections of metrics. We consider different machine learning methods including Decision Tree (DT) [80], Random Forest (RF) [157], Support Vector Machines (SVM) [65], and Logistic Regression (LR) [77]. The results show that learning models can be built using software metrics as input to determine when the pipeline has reached its maximum capacity in a given configuration. The comparison of different models shows that the ones based on the *reliable* metrics identified in Chapter 3 are the one that provides the best accuracy. Models that only rely on server-side metrics can also achieve very good results. Furthermore, our extensive analysis shows the robustness of the obtained models that can generalize to new setups, to new numbers of clients, and to both new setups and new numbers of clients.

Towards a general approach of bottleneck identification

To assess the applicability of our results to other multi-tier systems, in our last contribution (Chapter 5) we run the analytical study and evaluate the learning-based approach on a different multi-tier distributed system, namely the LAMP Web stack. Considering 8 different setups of this stack, the results confirm the main findings we obtained through the study of the data processing pipeline. There are always reliable metrics that can help decide when the server has reached its maximum capacity in the multi-tier system. Collecting a few metrics of different metrics types is sufficient to identify the limiting component for most tested setups. Machine learning models based on a set of *reliable* metrics can accurately identify when the server has reached its maximum capacity, and such models are robust to significant changes in the configuration and the workload of multi-tier systems

1.4 Outline

This thesis is organized as follows. Chapter 2 gives some background on distributed systems and performance bottlenecks. Chapter 3 presents our first contribution, the analytical study for identifying reliable indicators of performance bottlenecks in the data processing pipeline. Chapter 4 presents our second contribution where we study the use of machine learning techniques to build models that can identify the limiting component in the data processing pipeline. Chapter 5 presents our third contribution where we apply the two approaches to a Web stack. Finally, Chapter 6 concludes this thesis and discusses future research directions that we believe are worth investigating.

Chapter 2

Background

Contents

This chapter provides the necessary background related to the other chapters of this thesis. In Section 2.1, we first give a primer of distributed systems and the main aspects that influence their design including the hardware aspects, the different architectural styles, and the features that they can provide through replication and partitioning. We discuss the performance concepts of distributed systems and present the major performance metrics as well as the different types of performance issues in Section 2.2. Finally, in Section 2.3, we present in detail the data processing pipeline as our case study. We describe the architectural design of the full data processing pipeline including its tiers/components. For each tier/component, we present the main design principles, provided features, factors that affect its performance, the role of the component and how it contributes to the studied system. We also present the importance of these components and how widely they are in use.

2.1 Distributed applications are complex

In this section, we present what a distributed system is, we talk about different architectural styles of distributed systems and how these different styles contribute to the complexity of these systems. The

complexity of distributed systems is implied by different factors. We classify these factors into (i) Architecture and interactions; (ii) Complexity of the hardware infrastructure; (iii) Partitioning and replication aspects.

2.1.1 What is a distributed system?

Several definitions of distributed systems have been introduced in the literature [91,142]. Tanenbaum et Van Steen [142] define a distributed system as following:
A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.
This definition refers to two characteristic features of distributed systems. The first one is that a distributed system is a collection of computing elements each being able to behave independently of each other. A computing element (generally a node), can be either a hardware device or a software process. A second feature is that users (can be people or applications) believe they are dealing with a single system. This means the autonomous nodes need to collaborate in one way or another.

Moreover, as technology has evolved significantly in the last decades, distributed systems become more complex. The complexity emerged from using both developed software technologies and hardware infrastructure. In other words, distributed applications are composed of many software components that provide different functionalities and that interact with each other. This complexity makes understanding the way systems perform, to improve their performance, a non-trivial task.

2.1.2 Architecture and interactions in distributed systems

The system architectural style refers to the way that the system components are connected, the data exchanged between the components and the way those components are configured. As mentioned before, a system component refers to the software building block in the system (e.g., Kafka [22] is a system component in a data processing pipeline (for more details see Section 2.3)).

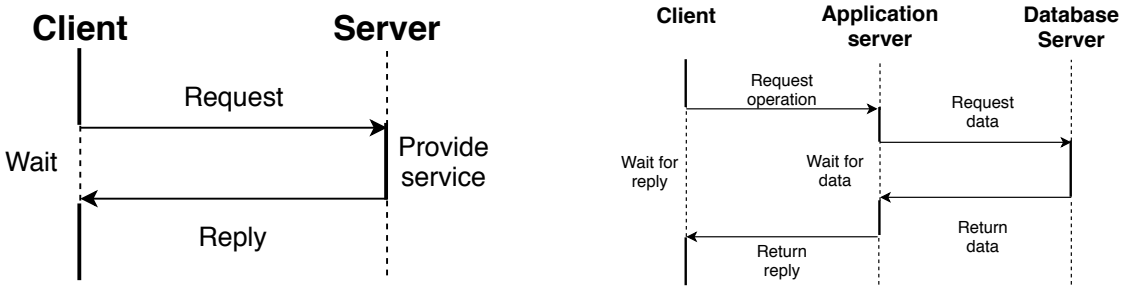
Thus, distributed systems can be designed in one or another way based on different aspects. We describe these architectural aspects in the following.

2.1.2.1 Monolithic, multi-tier, and service-oriented architectures

The monolithic architecture in the software engineering context describes a single-tier software application in which the user interface and data access code are combined into a single program from a single platform [31]. The design philosophy is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function. In other words, a system that follows a monolithic architecture looks like one piece.

On the other hand, a system might be designed in a multi-tier fashion in which presentation, application processing, and data management functions are physically separated [99]. Figure 2.1 (b) shows a three-tier architecture comprising: client, application server, and database server. In this architecture, a client requesting the application server which in turn requesting the database server. The database server replies with the request data to the application server which in turn sends it to the requested client. The server application acts as a client when it is requesting data from the database server. A typical example of the three-tier architecture usage is on web sites. In this case, we have three tiers: the first tier is a web server which acts as an entry point to the website, passing the client requests to the second tier, the application server where the actual processing is done. The application server also in its turn interacts with the third tier which is the database server, asking for the requested data.

In the context of the multi-tier architecture, a client-server model represents a single-tier architecture where we distinguish between two groups of actors. A *server* is a process implementing and managing services, for example, a file system service or a database service. A *client* is a process that requests a service from the server. The client-server architecture is also known as a *request-reply model* [105]. Each of the clients and the server represents a system component in the client-server architecture. This type of architecture has one or more client computers connected to a central server over a network or internet connection. Figure 2.1 (a) shows a general client-server interaction model where a client is requesting a service that is implemented in the server. The client sends the request to the server and waits for a reply.



(a) Client-server architecture.

(b) Multi-tier architecture.

Figure 2.1 – Common distributed systems architectural styles [142].

Service-oriented is a generalization of the previous model where more complex interactions between services can occur (not necessarily a pipeline) [118]. The numerous and complex interactions between components of a distributed system make it difficult to analyze and optimize its performance. In multi-tier systems, the interaction between the system components is more complex than in a system with two components where one of those components is limiting the system. While in multi-tier systems, issues may originate from complex interactions between system components that perform well when analyzed in isolation. This can occur due to a cascading failure when one failed component causes performance issues in others. To fix the resulting issue, we must analyze the relationships between components and understand how they interact.

2.1.2.2 Master-slave and peer-to-peer architectures

Another characterization of distributed systems architectures is based on the role of the system components (i.e., asymmetric vs. symmetric) [142]. This aspect leads to two main architectures: (i) Master-slave architecture, where the roles of the system components are asymmetric; (ii) Peer-to-peer architecture, where the roles of system components are symmetric. In a master-slave architecture, all nodes[1] are unequal (i.e., there is a hierarchy). The master acts as a central coordinator where all decisions can be made. On the other hand, in a peer-to-peer (P2P) architecture, all nodes are equal (i.e., there is no hierarchy). There is no central coordinator which may allow the system to better scale at the cost of harder decision making.

2.1.2.3 Communication models

The way the distributed system components (tiers) communicate contributes to defining the design style that the system follows. In this context, communications can be categorized along two axes: (i) Point-to-point vs. one-to-many; (ii) Synchronous vs. asynchronous.

[1] We use term node to refer to a machine.

Point-to-point vs. one-to-many

Point-to-point communication is the basic message passing communication model that provides two main primitives: *send* and *receive*. In other words, point-to-point communication involves a pair of processes, one process sends a message while the other process receives it (e.g., TCP sockets and RPC) [97]. This is contrasted with a one-to-many communication model. This model is for instance implemented by publish-subscribe systems where a message is transferred from a sender to multiple receivers.

Synchronous vs. asynchronous

In synchronous communication, multiple parties are participating at the same time and wait for replies from each other. In other words, the sender is blocked until its request is known to be accepted by the receiver. Typically, client-server interactions reply on this model. In asynchronous communication, a publisher/sender continues sending messages without waiting for the final destination to receive it [142]. Modern distributed applications include all these kinds of communication patterns. It can make applications performance difficult to analyze as issues may appear in very different ways depending on the model.

2.1.3 Complex hardware infrastructure

Distributed infrastructures are complex. Many characteristics of modern distributed infrastructures can influence the design and the performance of distributed applications. We present some major points here:

- Network: The first aspect to consider for the network is the distance between the computing resources. If the resources are geographically distributed and connected through a wide-area network (WAN), the latency between nodes is going to be much higher and the bandwidth much lower, than if all the nodes are in the same local-area network (LAN). Inside a LAN, several network topologies exist (mesh, ring, trees, etc.), each having advantages and drawbacks in terms of performance and reliability. In this thesis, we consider nodes interconnected through a LAN network organized as a hierarchical tree topology, a very common basic topology in data centers [72].

- Multi-core processors: With the end of Dennard scaling, processors have evolved from powerful single-core processors to complex multi-core architectures. Modern multi-core processors comprise multiple computing elements that can collaborate through shared memory and a complex hierarchy of private and shared caches [141]. Parallelism can be further increased by using techniques such as simultaneous multithreading [143]. Also, to increase the capacity of servers, multiple processors can be integrated inside the same node. Such NUMA architectures make it even more challenging to optimize application performance [115]. Such complex processor architectures are considered in this thesis.

The emergence of complex hardware infrastructures has led to complex software to fully take advantage of the underlying resources. Today's distributed software is highly configurable with many configuration parameters that can be tuned to best adapt to the underlying infrastructure and the workload [130]. Yet, it is not trivial to configure the software in a way that it is effectively benefiting from the underlying hardware [70, 152, 153, 155]. All these levels of complexity have made difficult to spot performance issues and to identify the causes of these issues.

Also, the emergence of cloud computing and virtualization technologies has an impact on designing, deploying, and optimizing distributed systems as these technologies involve special hardware, software, or a combination. However, we do not consider cloud computing and virtualization in this thesis. We focus

on systems deployed directly on top of the hardware where performances are not impacted by layers of virtualization.

2.1.4 Partitioning and replication

partitioning and replication are two major techniques used by distributed software at scale to improve performance and reliability. We introduce these two techniques below:

2.1.4.1 Partitioning

Partitions are defined in such a way that each piece of data (record, row, or document) belongs to exactly one partition [113]. The main reason for wanting to partition data is scalability. System scalability is a fundamental term that has been introduced in distributed systems as it affects their design and thus their performance. A system is described as *scalable* if it remains effective when there is a significant increase in the number of resources and the number of users [91].

Thus, partitioning in distributed systems implies that each component in the system can be divided into several partitions spread on separate machines for parallel processing. In other words, partitioning involves taking a system component (or data), splitting it into smaller parts, and subsequently spreading those parts across the system.

Furthermore, partitioning involves that messages are exchanged between instances within the same component. Also, more partitions imply synchronization and exchanging messages between the systems partitions on one hand. On the other hand, more partitions require applying a load balancing strategy for selecting partitions to store messages.

2.1.4.2 Replication

In addition to the partitioning aspect, each component/partition in the system typically has several replicas (instances) for fault tolerance and high availability purposes. Replication plays an important role in distributed systems [113]. Replicating a software component and/or data is necessary for several reasons: (i) to keep data geographically close to end-users (and thus reduce latency), (ii) to allow the system to continue working even if some parts of the system have failed (and thus increase availability/fault tolerance), and (iii) to scale out the number of machines that can serve read queries (and thus increase read throughput).

Replication requires complex interactions between the replicated components and all the clients to ensure some level of consistency in the operations that are run on the data.

2.2 Distributed systems performance

Gregg [102] defines the performance of a system as a study of the entire system, including all hardware and software components in the system. Users spend significant both time and effort understanding systems so they can tune them for better performance.

2.2.1 Performance metrics

Several metrics on different levels of a distributed system including hardware resources and software components have been used to measure its performance. Hence, we distinguish between two main categories as follows:

- Application-level performance metrics: This includes two main metrics:
  - The *throughput* refers to the number of messages/requests that the system can process in time unit. In other words, system throughput is the rate of work performed. For example, in database contexts, the throughput may refer to the transactions performed per second.
  - The *latency (or response time)* refers to the amount of time a system needs to process a message/request and send back a reply to the client. Latency, also, can mean the time for any operation (e.g., a database query or an application request) to complete [102].
- System-level performance metrics: This includes *resource usage* metrics. As explained in several works [102, 117, 126], the saturation of some system resources often causes undesirable performance effects and allows diagnosing performance bottlenecks. Also, the resource consumption metrics are important and can give a hint of what is limiting system performance [111, 162].

2.2.2 Performance issues

As we have mentioned earlier, systems have become more complex as it is comprising many software components that provide different functionalities as well as interact with each other. On the other hand, the complexity of the hardware also plays a part, adding complexity to understand systems and their performance. Thus performance issues have been raised and categorized into two main classes:

- Performance anomaly: It refers to patterns in data that do not comply with the normal/expected behavior [83]. Figure 2.2 illustrates an example of throughput anomaly. The group of points P represents a short dip in system throughput. However, in this thesis, we study the *global throughput* of the system. We assume that the observed system has a steady-state behavior and that the performance-limiting component is stable over time. In other words, we do not consider setups exhibiting transient or rapidly alternating bottlenecks [148, 149]. We define the *global throughput* of the system by the amount of work (or tasks) that the system can handle per time unit (see Section 3.3 for more details).
- Performance bottleneck: It is a resource or an application component that limits the performance of a system [102]. Malkowski et al. [122] describe a bottleneck component as a potential root cause of undesirable performance behavior caused by a limitation (e.g., saturation) of some major system resources associated with the component [116]. We distinguish between two main types of performance bottlenecks:
  - **Resource saturation bottlenecks**: A resource is saturated when its capacity is fully utilized or past a set threshold [102]. According to Gregg [102], saturation may also be estimated in terms of the length of a resource queue of jobs or request to be served by that resource. For example, a CPU near 100% utilization may result in a congested queue and growing latency.
  - **Software misconfiguration bottlenecks**: The software might exhibit performance issues as it is not well configured. For example, in a Kafka cluster [22] where Kafka clients send messages to be written into Kafka brokers, the performance of the system might reach its maximum even with no obvious saturated hardware resources (while increasing the number of clients). In this case, a bottleneck appears to be on the broker side even if its resources are underutilized. However, tuning the broker configuration by, for example, increasing the number of threads that are responsible

for receiving client messages, effectively improves the overall performance of the system. In the example just mentioned above, a bottleneck appears as a consequence of misconfigured software.

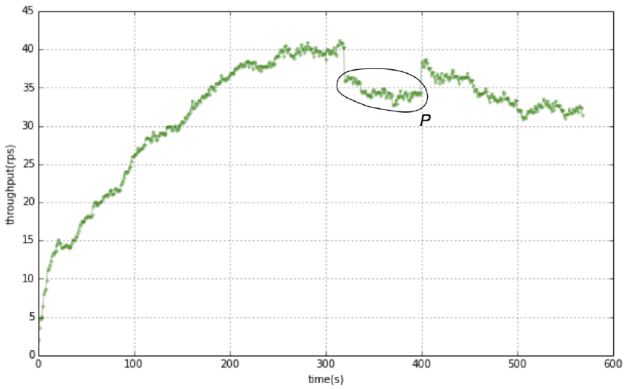


Figure 2.2 – Illustration of throughput anomaly [106].

2.2.3 Investigating performance bottlenecks

In this thesis, we focus on performance bottlenecks. We study different use cases of distributed components composing complex applications. The studies cases comprise: (i) single-tier systems, and (ii) multi-tier systems. For each studied case, we make the distinction between (i) the client side and (ii) the server side. The client side represents the system component that is responsible for injecting the data into the system. The server side represents the system component that encompasses all other system components that participate in processing the data injected in the system. The server-side can include one or multiple components.

For the single-tier systems, we strive to answer the question concerning if the server reaches its maximum capacity and thus, we consider the server as the limiting component in the system. In a case of multi-tier systems, besides, to find when the server reaches its maximum capacity, we strive to identify the tier that is limiting the system performance.

2.3 Data processing pipeline as a case study

The main focus of this thesis is the study of performance bottleneck in data processing pipelines. In this section, we present in detail the studied data processing pipeline. We describe the architectural design of the full data processing pipeline including its tiers/components. For each tier/component, we present the main design principles, provided features, factors that affect their performance, the role of each component and how it participates in the pipeline. We give examples from the production of where and how these software components are used. Then, for each component in the pipeline, we present the main performance metrics on both the software and the hardware levels.

2.3.1 Preface

The classic approach to data processing pipeline is to write a program that reads in data, applies a transformation (function) on it, and outputs new data [75,76]. There is a wide variety of pipeline applications. A pipeline can involve multiple stages; each stage is a separate process with dependencies on other stages.

A common example of a data processing pipeline is the Extract Transform Load processing (ETL) [76], where the data is extracted from a single (or multiple) sources, transformed, and then loaded (or written) into another data source. Typically, an ETL pipeline prepares the data for further analysis or serving. When used correctly, ETL pipelines can perform complex data manipulations and ultimately increase the efficiency of the system. Some examples of ETL pipelines include (i) preprocessing steps for machine learning or business intelligence use cases, (ii) computations, such as counting how many times a given event type occurs within a specific time interval, (iii) calculating and preparing billing reports, and (iv) indexing pipelines like the ones that power Google's web search.

In this study, we consider a data processing pipeline comprising Kafka [22], a distributed publish-subscribe messaging system, Spark Streaming [30], a Big Data processing and analysis framework, and Cassandra [15], a distributed NoSQL database management system. Furthermore, we assume that ZooKeeper [52] is used as a coordination service. We build data processing pipeline applications using these software components. In these applications, a client publishes the messages to Kafka where it is stored. Spark Streaming integrates with Kafka to consume the injected messages. Spark Streaming also integrates with Cassandra where the results of the processed messages are stored. Figure 2.3 illustrates the three main tiers in the data processing pipeline. The first tier represents the data ingestion tier, the second tier represents the data processing and analyzing tier, and the last tier represents the database where the results of the processed messages are stored. Also, as we mentioned before, a coordination service is provided by Apache Zookeeper which is deployed together with Apache Kafka.

The above-described software stack is widely used in production for analyzing streams of data. For example, Apache Kafka is used at LinkedIn [9] for managing activity stream data and operational metrics. This powers various products like LinkedIn Newsfeed, LinkedIn Today in addition to their offline analytics systems. At Spotify [11], Kafka is used as a central component for its log delivery system. Their goal is to send all the data produced in all their hosts into their Hadoop cluster for later processing. By adopting Kafka as part of their pipeline they can reduce the average time needed to transfer logs from 4 hours to 10 seconds [27]. At Twitter [25], Kafka is used as part of their Storm stream processing infrastructure [53]. More details about using Apache Kafka in production are available here [36].

Regarding Spark Streaming, Databricks [28] provides a cloud-optimized platform to run Spark and machine learning applications on Amazon Web Services and Azure. eBay [4] uses Spark core for log transaction aggregation and analytics. More details about the usage of Apache Spark in production are available here [47].

Apache Cassandra is used in production as well by many companies that have large and active data sets [15]. Companies like eBay [4], Instagram, and Netflix [7], etc., [15] are using Apache Cassandra. eBay uses Cassandra for Social Signals on eBay product and item pages as Cassandra provides needed features like scalable counters, real (or near) time analytics on collected social data and good write performance [26]. Netflix [7] migrates from Data center Oracle to Global Apache Cassandra [24] as it is a distributed key-value store and it provides the required scalability and high availability.

Furthermore, combining these components is, also, in high use in practice. For example, combining Apache Kafka and Spark Streaming provides a centralized *client even* ingestion point for the Royal Bank of Canada's internal systems through either a web service or text file daily batch feed [49]. A full data pipeline comprising, Apache Kafka, Spark Streaming, and Apache Cassandra, is used for real-time recommendation at Netflix [46] where they migrate from the traditional recommendations which are precomputed in a batch processing fashion into real-time recommendations using Apache Spark Streaming. This helped them in leveraging real-time data for model training, such as providing the right personalized videos in a member's

Billboard and choosing the right personalized image soon after the launch of the show. A combination of Apache Kafka, Spark Streaming, and HBase [16] is used at Airbnb [35, 45]. Moreover, nowadays, Spark, Mesos, Akka [3], Cassandra, and Kafka (SMACK) has become the foundation for big data applications [96].

In addition to the usage of these components in production, it is worth mentioning that building a data processing pipeline is not exclusive to these components. For example, RabbitMQ [13], Apache ActiveMQ [12] and others [51] can replace Apache Kafka as a messaging system. Apache Storm [23], Amazon Kinesis [10], Apache Flink [20], and others [48] can replace Spark as a big data processing and analysis framework. Regarding the alternatives to Apache Cassandra, PostgreSQL [6], MongoDB [18], and others [50] are possible options.

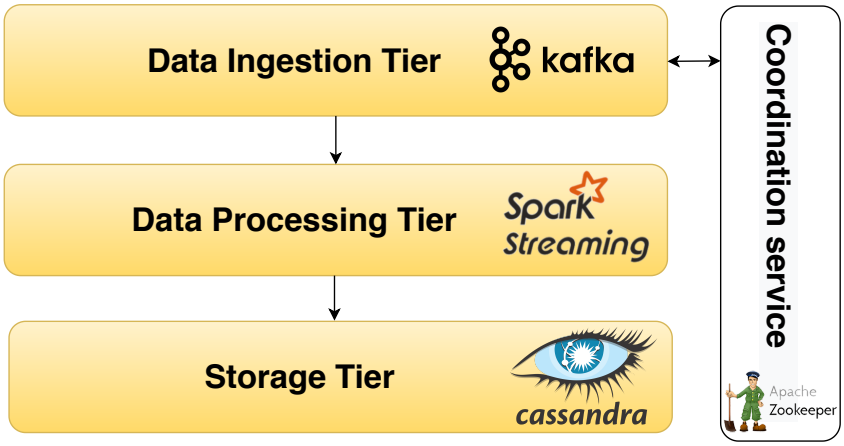


Figure 2.3 – Data processing pipeline

In the following sections, we present in detail each component of the data processing pipeline.

2.3.2 Data processing pipeline components

2.3.2.1 Apache Kafka

Apache Kafka [22] is a publish-subscribe messaging system. Publish/subscribe messaging is a pattern that is characterized by a sender (publisher) that sends a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the messages somehow, and the receivers (subscribers) subscribe to receive certain classes of messages. Publish-subscribe systems often have a broker, a central point where messages are published, to facilitate classifying and receiving messages.

Kafka relies on three sets of actors: producers, brokers, and consumers. Figure 2.4 illustrates a general Kafka architecture with the main actors. The producer interface allows an application to publish a stream of records/messages to one or more Kafka topics stored in the brokers. A Kafka topic is the container with which messages are associated. The consumer interface allows an application to subscribe to one or more topics and to receive the stream of records from the brokers.

Topics are partitioned across multiple machines/nodes and the partitions are replicated and distributed for high availability. Each partition has one server that acts as a *leader* and zero or more servers that act as *followers*. The leader handles all read and write requests for the partition while the followers replicate the leader. Kafka always allows consumers to read-only from the leader partition. A leader and follower of a partition should never reside on the same broker for obvious reasons. Followers are always synchronized with

a leader. If the leader fails, one of the followers automatically becomes the new leader. Each server acts as a leader for some of its partitions and as a follower for others so the load is well balanced within the cluster [114]. The communication between a consumer and a broker is asynchronous point-to-point communication. When considering the messages transferred from the producers to the consumers, communication can be defined as one-to-many and asynchronous.

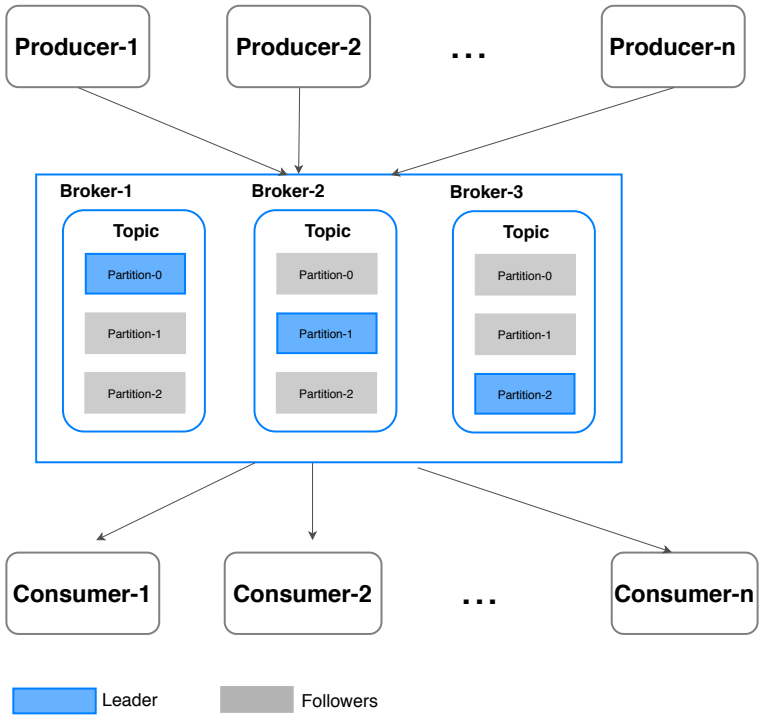


Figure 2.4 – Kafka architecture

Apache Kafka uses Zookeeper [52] to store metadata about the Kafka cluster, as well as consumer client details. Essentially, ZooKeeper is a centralized service for maintaining configuration information, naming, and providing distributed synchronization.

Kafka's performance can be impacted by many factors including message size, compression, and the number of partitions/topics, etc. For instance, for a messaging system, small messages can cause a performance issue as they magnify the overhead of the bookkeeping the system does. Hence, generally, in a Kafka cluster, increasing the size of the messages increases the Kafka throughput (it also leads to a higher latency). Also, the number of partitions (where messages are written) forms the unit of parallelism in Kafka and has an important impact on its performance. In general, in a Kafka cluster the more partitions there are, the higher the throughput one can achieve[2]. On the other hand, with multiple partitions, Kafka does not guarantee a global ordering of the messages.

Understanding the performance of Kafka taking into account all the parameters that can be configured is challenging. In this thesis, we examine different setups (see Table 3.10) of Kafka clusters as we describe in Chapter 3.

Moreover, the complex architectural style that Kafka follows makes understanding its performance as a non-trivial task. Kafka uses a peer-to-peer configuration of brokers where Kafka clients (producers) directly control how a particular piece of data is assigned to a particular partition. On the other hand, Kafka, also,

[2]Note that in some cases, having too many partitions may also have negative impact. For example, more partitions requires dealing with more open file handlers in the broker. Also, more partitions may increase the end-to-end latency which is defined by the time from when a message is published by the producer to when the message is read by the consumer [32].

follows a master-slave architecture style in its implementation of partitioning. Kafka topics are split into partitions. A partition lives on a physical node and persists the messages it receives. A partition can be replicated onto other nodes in a master-slave relationship. There is only one *leader* node for a given partition that accepts all reads and writes – in case of failure a new leader is chosen. The other nodes just replicate messages from the *leader* to ensure fault-tolerance [41, 95].

Therefore, setting that many performance parameters and understanding the complex architecture of a Kafka cluster, makes it a difficult task to explore and identify its performance issues.

2.3.2.2 Apache Spark

In this section, we first present the main features that Spark provides, then we explain the fundamental components that form Spark architecture. Finally, we explain the main data structures in Spark and the kind of operations that can be applied to it.

Spark features

Apache Spark [29] is a unified analytics engine for big data processing. Spark includes five main components: Spark Core, Spark Streaming, Spark SQL, MLlib, and GraphX (see Figure 2.5). Spark supports several cluster managers including Apache Mesos [110] – a general cluster manager that can also run Hadoop MapReduce [21] and service applications, and YARN [145] – a resource Manager that has (i) a scheduler that allocates resource to the various running application, and (ii) an application manager that manages applications across all the nodes in the system. In this thesis, we use the Spark Standalone manager – a simple cluster manager included with Spark that makes it easy to set up a cluster.

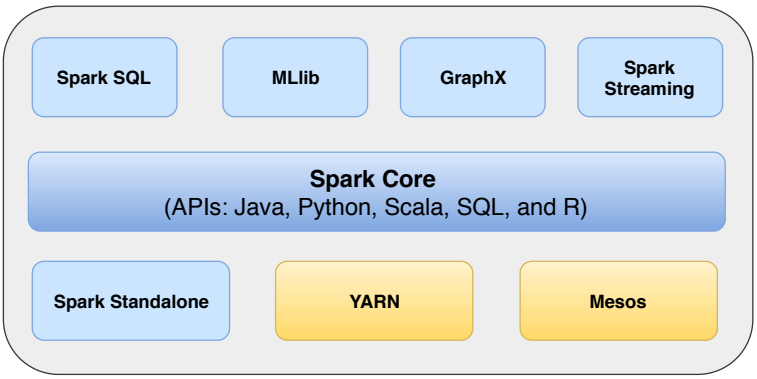


Figure 2.5 – Apache Spark ecosystem

Spark Core [29] is the foundation of the overall Spark project. All the functionalities being provided by Apache Spark are built on the top of Spark Core. Spark Core provides distributed task dispatching, scheduling, and basic I/O functionalities. Also, it delivers speed by providing in-memory computation capabilities.

Spark has stream-processing capabilities [30], which make Spark able to handle real-time data. Spark uses micro-batching[3] for real-time streaming. Besides, Spark Streaming adds specific streaming operators: windowing (the grouping of records from past time intervals on a sliding window), incremental aggregation over a sliding window, state tracking (to transform a list of events into a list of updated states). Spark can access data from sources like Kafka, Flume [104], or TCP sockets. Spark Streaming implements

[3] Micro-batching is a technique that allows a process/task to treat a stream of data as a sequence of small batches.

micro-batching through the Discretized Streams (D-Streams) abstraction [156]. In Discretized Streams, incoming data is grouped in an RDD and the lineage graph[4] associated with it is computed repeatedly, for example, every second.

Spark SQL [69] is a component on top of Spark Core that introduced a data abstraction called DataFrames, which is a Dataset organized into named columns. A DataFrame is conceptually equivalent to a table in a relational database [29] on which operations such as select, join and group-by can be applied.

MLlib [125] is another component of Spark. It is a scalable machine learning library, which implements various machine learning algorithms on top of DataFrames.

Finally, GraphX [101] is a graph analytics engine and data store.

In thesis, we are interested in Spark Streaming as we examine the case of the data streaming processing pipeline.

Spark architecture

Spark architecture encompasses three main components: the driver, the cluster manager, and the executors (see Figure 2.6).

The driver Spark driver program hosts the `SparkContext` which is the coordinator for a Spark application. Specifically, to run on a cluster, the `SparkContext` can connect to several types of cluster managers (either Spark's standalone cluster manager, Mesos or YARN), which allocate resources across applications. Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for the application. Next, it sends the application code (defined by JAR or Python files passed to `SparkContext`) to the executors. Finally, `SparkContext` sends tasks to the executors to run. Spark is based on *staging* which is a set of parallel tasks. In other words, each job that gets divided into smaller sets of tasks is a stage. A spark stage can be associated with many other dependent parent stages. However, it can only work on the partitions of a single RDD. Also, the boundaries of a stage in spark are defined by shuffle dependencies.

The cluster Manager The cluster manager is an optional component, which is only necessary if Spark is executed in a distributed way. It is responsible for administering the machines that will be used as workers. As just mentioned above that there are several types of cluster managers. In this study, we use the *Standalone* one which is a simple cluster manager included with Spark that makes it easy to set up a cluster [29].

The executors Spark executors are worker nodes' processes in charge of running individual tasks in a given Spark job. A spark job is a parallel computation consisting of multiple tasks that get spawned in response to a Spark action [29]. Spark executors are launched at the beginning of a Spark application and typically run for the entire lifetime of an application. When the `SparkContext` is created, each worker starts an executor as a separate process (Java virtual machine (JVM)), and it loads the application code, too. The executors connect back to the driver program and the driver assigns them transformations to be executed on some data partitions.

[4] RDD Lineage is a graph of all the parent RDDs of an RDD

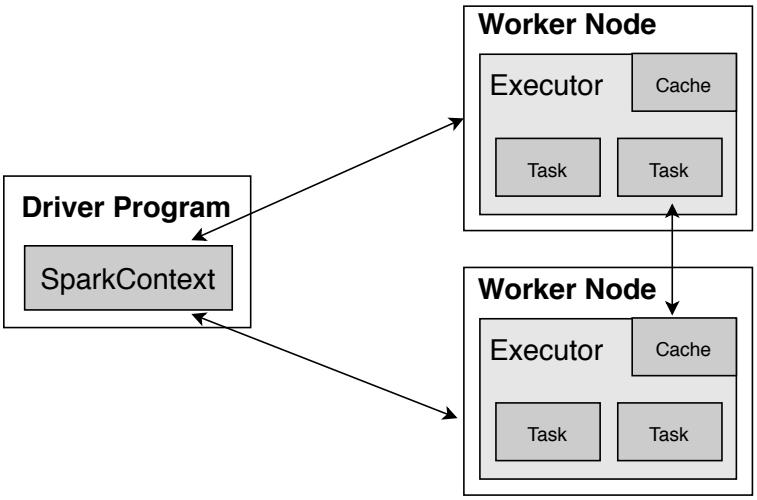


Figure 2.6 – Spark cluster architecture [29].

Spark internals

The fundamental data structures in Spark is *Resilient Distributed Datasets (RDDs)*. They are at the core of Spark and built from input data, and on which one can apply many operations. RDDs are an immutable collection of objects which are distributed on different nodes of the Spark cluster. Each Spark RDD is partitioned across servers so that transformations can be executed on different nodes of the cluster in parallel. In this context, it is worth mentioning that some transformations imply RDD shuffling which is a process of redistributing data across partitions that may or may not cause moving data across JVM processes or even over the network between executors on separate nodes. Note that shuffling might hurt the performance of a Spark cluster as it involves a large number of disk IO, serialization, network data transmission, and other operations [29, 44].

Operations that can be performed on RDDs are:

- Transformations: Transformations produce a new RDD from existing RDDs. For example, `map` and `filter` transformations.
- Actions: Actions return an output that is not an RDD. Applying an action on an RDD triggers the executions of all preceding transformations. Examples of actions on RDDs are `count` and `collect`.

In Spark, all the dependencies between the RDDs are logged in a graph. This is called a lineage graph. Essentially, the evaluation of an RDD is lazy. It means that when a series of transformations are performed on an RDD, they are evaluated immediately. Only the execution of actions triggers the execution of all transformations included in the lineage of the RDD.

Like for many other systems, many parameters can affect the performance of Spark (e.g., batch interval[5], and the number of Spark executors). Thus, setting these parameters impacts the performance of a Spark cluster. On the other hand, it is complex to get performance out of Spark and this is due to (i) the complex communication between Spark workers, also between the workers and the driver, and (ii) the in-memory nature of most Spark computations. Spark programs can be bottlenecked by any resource in the cluster: CPU, network bandwidth, or memory.

[5]The batch interval defines the size of the batch in seconds. For example, a batch interval of 5 seconds will cause Spark to collect 5 seconds worth of data to process.

Besides, Spark has a master-slave architecture, where one central coordinator (the Driver) communicates with many distributed workers nodes (executors), and also can be a performance limiter.

Therefore, combining all factors just mentioned above requires a non-trivial effort to understand and troubleshoot performance issues in a Spark cluster.

2.3.2.3 Apache Cassandra

Cassandra [15] is a distributed NoSQL database management system. NoSQL is a general name for the collection of databases that do not use Structured Query Language (SQL) or a relational data model [103]. Some of the salient features of NoSQL databases are that they can handle extremely large amounts of data, can be replicated easily, and they are practically schema-free. Cassandra is designed to handle big data workloads across multiple machines/nodes with no single point of failure [103].

Cassandra is a partitioned row store database, where rows are organized into tables with a required primary key. Cassandra's architecture allows any authorized user to connect to any node in any data center and access data using the CQL (Cassandra Query Language) language. For ease of use, CQL uses a similar syntax to SQL and works with table data.

Figure 2.7 illustrates the high-level architecture of Cassandra [15]. This architecture encompasses three main components:

- Node: It is the basic infrastructure component of Cassandra where data is stored.
- Data center: It is a collection of related nodes. A data center can be a physical data center or virtual data center. Different workloads should use separate data centers, either physical or virtual. Using separate data centers prevents Cassandra transactions from being impacted by other workloads and keeps related data close to each other for lower latency. Depending on the replication factor, data can be written to multiple data centers.
- Cluster: It contains one or more data centers. It can span multiple physical locations.

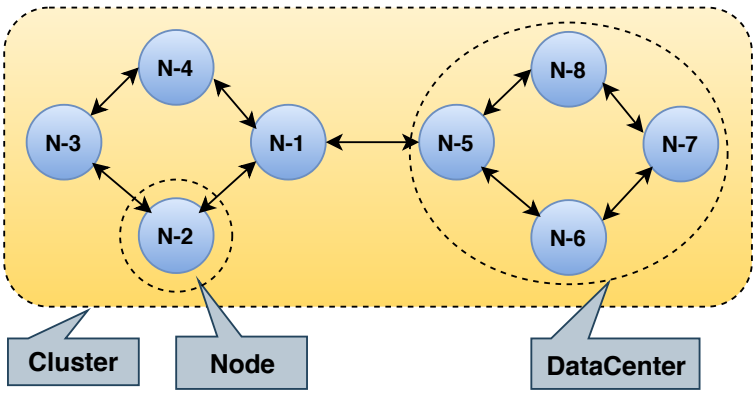


Figure 2.7 – A high-level architecture of Cassandra

Each node in a Cassandra cluster frequently exchanges state information about itself and other nodes across the cluster using a peer-to-peer gossip communication protocol. A sequentially written commit log on each node captures write activity to ensure data durability. Data is then indexed and written to an in-memory structure, called a *memtable*, which resembles a write-back cache. Each time the memory structure is full, the data is written to disk in an *SSTables* data file [113]. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates *SSTables* using a process called *compaction*,

discarding obsolete data marked for deletion. To ensure all data across the cluster stays consistent, various repair mechanisms are employed.

Regarding *read* operation, a coordinator node, which is a Cassandra node that is responsible for managing the entire request path and to respond to the client [39], sends a direct request to one of the replicas. After that, the coordinator sends the digest request to the number of replicas specified by the consistency level and checks whether the returned data is updated. After that, the coordinator sends a digest request to all the remaining replicas. If any node gives out of date value, a background read repair request will update that data. This process is called the read repair mechanism. The Cassandra consistency level is defined as the minimum number of Cassandra nodes that must acknowledge a read or write operation before the operation can be considered successful.

In this study, we deploy a Cassandra cluster in one physical data center. Some nodes in the cluster are dedicated to hosting Cassandra instances. To ensure reliability and fault tolerance, data replicas are stored on multiple nodes (in our case replication factor equals to 3). All these settings described in this section (number of nodes, replication factor, consistency level, and the architectural design) impact the performance of a Cassandra cluster and its troubleshooting.

2.4 Conclusion

In this chapter, we have presented the required background knowledge of the later chapters of this thesis. We have seen that distributed systems are complex. Their complexity comes from many factors including the fact that they are composed of a variety of software components that provide different functionalities, and that have many interactions involving both control messages and data. From the architectural point of view, many differences can also be noticed between single-tier and multi-tier systems; peer-to-peer and master-slave architectures; point-to-point and one-to-many communication models. Also, the complex architecture of the hardware infrastructure adds more complexity to the design and the deployment of these systems.
Systems may have performance issues that might originate from a high utilization of the hardware resources or the software misconfiguration. By system performance, we refer to the system throughput measured as the number of messages/requests that the system can process per time unit. We point out that all the factors we have mentioned above make it a challenging task to identify if a given system is under-performing and, if it is the case, pinpoint the bottleneck component in that system. In this thesis, we consider a widely used data processing pipeline as a studied case. All the components included in this pipeline are widely used by companies and academia.

In the following chapter (Chapter 3), we present our analytical study on the data processing pipeline. We describe a methodology that aims at finding a set of key metrics that can be used as *reliable* and *general* indicators of performance bottlenecks in the data processing pipeline. By *reliable*, we mean that the indicator should provide accurate results despite significant changes in configuration setups. By *general*, we mean that the indicator should apply to different kinds of application components.

Chapter 3

Analytical Study to Identify Reliable Indicators of Performance Bottlenecks

Contents

In this chapter, we present our analytical study to identify reliable indicators of performance bottlenecks in the data processing pipeline. The intuitive approach to identify performance bottlenecks in distributed systems is to monitor the system-level metrics (i.e., CPU, memory, network, and disk IO) and the bottleneck is in the component that relates to the saturated hardware resources. In this chapter, we show that:

- Relying on intuitive metrics like resource consumption metrics is not sufficient in identifying the limiting components in the data processing pipeline.
- We insist on the fact that there are hundreds of metrics exported by data processing pipelines and thus we strive to answer the following questions:
  - *Are there metrics exported by the data processing pipeline that we can rely on in identifying performance bottlenecks in the system?*
  - *How fine-grained are the conclusions that we can draw from these metrics i.e., can we just conclude if the bottleneck is on the client side or on the server side? Or can we also conclude, in the case where the server is composed of multiple components about, (i.e., the server is composed of Kafka, Spark, and Cassandra) which component is limiting the performance (i.e., is it Kafka, or Spark, or Cassandra)?*
- *In the case of the existence of these metrics, what are these metrics? How can we infer them? What are their characteristics?*

We start this chapter by presenting the related works that have been done on the topic of performance troubleshooting for distributed systems in Section 3.1. Section 3.2 describes the set of metrics that are studied for our analysis. We present in detail the steps that we follow to reply to the research questions mentioned above in Section 3.3. Section 3.4 demonstrates the proposed methodology through three concrete case studies of the data processing pipeline: a Kafka cluster, a Kafka/Spark Streaming cluster, and a Kafka/Spark Streaming/Cassandra cluster. Finally, we discuss our findings and we draw some conclusions in Section 3.9.

3.1 Related work

There has been much research devoted to performance troubleshooting for distributed systems [106,137,147]. Different aspects regarding performance troubleshooting and optimization have been explored at different levels. We present the related work in two main categories based on the question they are trying to answer: (i) Is the system performing efficiently? (ii) What are the performance issues and where do they lie? We also present the other works that do not necessarily fall into those two defined categories.

3.1.1 Is the system performing efficiently?

There has been much research [67,102,117,126,160] aimed at answering the following question: *Is the running system performing efficiently or not?*

Some research works try to reply to that question by examining the saturation of the system hardware resources (i.e., CPU, memory, and IO) [102,117,126]. The saturation of some system resources often causes undesirable performance effects and allows diagnosing performance bottlenecks. The saturated resource approach does thus consist in monitoring key resource indicators to dynamically detect overloaded components that could be performance bottlenecks. This approach is probably the most used as it is fairly easy to use in production systems. Several tools exist to collect performance metrics and report alerts when some of them reach a threshold [56, 102, 123]. Relying on resource consumption metrics is not sufficient for our problem as we show empirically that for all examined cases, none of the hardware resources reached its saturation or even a high usage (for more details see Section 3.4). Conversely, the absence of resource saturation is not necessarily a sign of fluid operation (e.g., the system may be excessively idle due to a cascade of blocking interactions with a slow thread).

CPI^2 [160] uses cycles-per-instruction (CPI) data obtained by hardware performance counters to identify slow tasks out of a set of independent but similar tasks running on a cluster of machines. To identify interfering tasks, CPI^2 looks for correlations between the CPI of the slow tasks and the CPU usage of the other tasks running on the same machine. CPI^2 then uses CPU hard capping to throttle the CPU consumption of interfering tasks and improve the performance of slow tasks. This technique is not sufficient for our problem, in which there are not only interferences between local tasks but also interdependencies (via synchronous and/or asynchronous interactions) between local and remote components.

Altman et al. present the WAIT tool [67], for pinpointing the causes of idle time in server applications. The tool provides a dashboard and a rule-based system for the business component of Java Enterprise applications, highlighting where threads wait (e.g., locks, database, network, disk). However, this tool only focuses on a single component, whereas, in our case, a holistic and complete view of all the components of an application might be necessary to fully understand and debug the system performance [74,102]. Performance issues may arise from complex interactions between system components that may perform efficiently while running in isolation. Solving performance issues in such systems often requires the whole system (both its internal and external interactions) to be investigated to gaining insights into the system activities and interactions.

3.1.2 What are the performance issues and where do they lie?

To fix performance issues in a system, it is necessary to identify the symptoms (*what?*) and also the root cause (*where?*). For example, we need to identify if CPU is limiting the performance of the deployed system, also what component is causing this bottleneck. In the context of data analytics frameworks, Ousterhout et al. [128,129] propose blocked time analysis, which allows quantifying how much faster a job would complete

if its tasks never blocked on disk or network I/O. Applying their methodology to Spark [29], they identify several causes for performance issues including high CPU utilization, garbage collection, and disk I/O. This approach is useful but currently limited to the scale/context of a single component (Spark), with specific instrumentation and assumptions regarding the execution model where there is a job comprising multiple tasks.

Pivot Tracing [120] is a recent monitoring framework for distributed systems that combines dynamic instrumentation with causal tracing. It correlates and groups system events across components and machines boundaries. As an example, this allows identifying the components that indirectly create resource contention downstream on other parts of the system. Pivot Tracing requires instrumentation provided by the developers or system users to propagate specific metadata. By design, Pivot Tracing is only aimed at pinpointing the root cause of known performance issues, not at detecting such issues.

Malkowski et al. [121] introduce a deterministic algorithm that automatically examines and analyzes the entire metric data derived from the pre-production configuration testing experiments. A limited set of interesting metrics is identified and ordered according to the degree of correlation with the high-level system performance (i.e., service level object (SLO)). Metrics that do not indicate a high resource saturation level are discarded. Given a known intervention (SLO begins to deteriorate), the authors identify all metrics that show evidence of a corresponding plateau (i.e., significant and permanent shift in average value) and a variability change in their first derivative (further evidence for a saturated resource). In other words, the metrics are selected based on predefined rules. The authors rely on building a simple performance model based on putting some assumptions together with observations from empirical data analysis. However, in our approach, we do not rely on some objectives (i.e., in [121] the authors focus on optimizing a specific objective function as the response time should stay stable with a given value) neither on a specific set of metrics that indicate a high resource saturation level. Instead, we perform an analytical study that aims at identifying reliable indicators, if they do exist, of performance bottlenecks. These metrics can be used in identifying the limiting component in the system.

3.1.3 Other existing work

There has been other research aiming at identifying bottlenecks in distributed systems using analysis. Wang et al. [148, 149] present an experimental study of rapidly alternating bottlenecks in n-tier web applications. Their study relies on using passive network packet tracing, which captures the arrival and departure time of each request of each server at microsecond granularity with a negligible impact on the servers. By correlating the throughput (i.e., request service rate) and the load (i.e., number of concurrent requests over a time interval or in other words, the number of completed requests in server in a fixed time interval, which can be 50ms, 100ms, or 1s) of each server of an n-tier application, their method is able to find short-lived alternating bottlenecks (lifetime of tens of milliseconds). However, in our study, we aim at pinpointing the limiting component in the system where the system reaches its saturation (i.e., maximum capacity) through analyzing a set of metrics that show trend changes in their evolution when the bottleneck shifts from one side to the other. Moreover, we assume that the observed system has a steady-state behavior and that the performance-limiting component is stable over time; in other words, we do not consider setups exhibiting transient or rapidly alternating bottlenecks.

Chung et al. [86] present a framework for automated performance bottleneck detection. The proposed framework relies on analyzing the distribution of active CPU time during a run of the application. The users can specify customized rules for bottlenecks and use the rules to query the performance data. Each bottleneck

is classified into one (or more) of the following dimensions: CPU, Memory, I/O, MPI Communication. The pre-defined set of rules is essentially aimed at pinpointing saturated resources. Furthermore, it does not guide defining new rules, nor for setting the thresholds associated with the rules. Besides, the tool provides no methodology to define a specific hierarchy or workflow between the rules.

NAP [73] is a simple methodology for application-agnostic diagnostic and remediation of performance hot spots in elastic multi-tiered client/server applications, deployed as collections of black box Virtual Machines (VMs). NAP relies on listening to the TCP/IP traffic on the virtual network interfaces of the VMs comprising an application and analyzes the statistical properties of this traffic. From this analysis, which is application independent and transparent to the VMs, NAP identifies performance bottlenecks (i.e., components and resources) that might affect application performance and derives remediation decisions that are most likely to alleviate the application performance degradation. However, in addition to the fact that observing and understanding in-out traces of hundreds of black-box components without prior knowledge may yield high false-positive detections, monitoring traffic at TCP/IP level is not sufficient for our problem as a complete view that reflects interactions between system components is needed. Moreover, we show experimentally that a combination of different metrics types (not only relying on metrics at the network level) is necessary to identify performance bottlenecks in the data processing pipeline.

Other kinds of work [108,150] rely on building a system performance model for multi-tier Web applications or online services by trying to satisfy one metric which is the *response time*. Some of these work define the response time by including two elements: (i) request execution time and queuing time caused by resource competition and (ii) network delay due to blocking inter-component communications. The proposed approach in [150] does not measure network effects, instead, it just takes into account the 90th-percentile of the response time. The response time is calculated as the time the request leaves minus the time it entered the system. The works presented in [108,150] target optimizing a specific objective function which is the *response time*. Their system actively observes application performance and tunes the admission rate of each stage to attempt to meet a 90th-percentile response time target. However, in our case, we aim to identify when the system reaches its maximum capacity based on the observation of a key set of metrics that act as reliable indicators of performance bottleneck. Furthermore, we show later in this chapter that relying on *response time* metrics is not sufficient to identify performance bottleneck in the data processing pipeline use case.

Similarly to the saturated resource approach [102, 117, 126] and the response time approach [108, 150], where authors rely on monitoring and satisfying specific metrics, Zhou et al. [161] proposed the DAGOR overload control approach for micro-service architectures. DAGOR uses the average waiting time of requests in the pending queue to profile the load status of a server. They empirically set a threshold of the average request queuing time to indicate server overload. We search for a solution that does not require the user to set thresholds.

vPerfGuard [151] is an automated model-driven framework for application performance diagnosis in consolidated cloud environments. vPerfGuard leverages the rich telemetry collected from applications and systems in the cloud, and the power of statistical learning to (1) automatically identifying system metrics that are most predictive of application performance, and (2) adaptively detecting changes in the performance and potential shifts in the predictive metrics that may accompany such a change. The framework mainly focuses on detecting bottleneck on various resources (CPU, memory, disk I/O) contention scenarios that are caused by workload surges or *noisy neighbors*. vPerfGuard differs from our work in the way that the authors rely on resource metrics while we show later (Sections 3.5.3, 3.6.3, and 3.7.3) that resource metrics are never reliable indicators of performance bottleneck in the data processing pipeline use case.

All works presented in [73, 108, 150, 151, 161] mainly differ from our work by relying only on one type of

metric at a time i.e., waiting time [161] or response time [150]. They also target troubleshooting performance issues of applications in consolidated cloud environments which is out the scope of this study.

To summarize, we have seen many research works done in the performance bottleneck identification area. Some of these works focus on monitoring the saturation of the system hardware resources [102, 117], while others monitor one metric at a time, either in a single component [129] or for the whole system [73]. Some of these works aim at satisfying defined objectives, mainly in a consolidated cloud environment [108]. Furthermore, some work [121] apply statistical study on a set of performance metrics to identify performance issues. However, they choose these metrics based on their correlation with Service Level Objectives (SLO) violations and where they indicate a high resource saturation level [121]. Different works use different kinds of metrics (system and/or application), but each work leverages in general one kind of metrics [108, 151, 161].

In this chapter, we propose an analytical study that aims at identifying *reliable* and *general* indicators of performance bottlenecks in multi-tier distributed systems and, more specifically, in a data processing pipeline. The study should provide system users with guidance about how to apply the proposed methodology to their systems and it gives insights about the important indicators that help in pinpointing a bottleneck in their systems. We evaluate our proposed methodology onto three sub-systems of the data processing pipeline where hundreds of different metrics can be exported.

We present the metrics exported by each software component in the data processing pipeline in Section 3.2. We detail the proposed approach in Section 3.3. The evaluation of the approach is in Sections 3.5, 3.6, 3.7.

3.2 Exported metrics

In this section, we present the set of metrics that are studied for our analysis. We present the metrics that are exported at the *software level* for each component of the data processing pipeline, as well as, the metrics that are exported at the *hardware level* to monitor resource consumption.

Each component in the data processing pipeline exports an enormous number of metrics on the software level. Trying to make sense of the system behavior from such a large number of metrics is very difficult. Identifying a small set of relevant and reliable metrics is not trivial either.

As a first step, we filter the set of metrics, retaining only the ones that are related to the main software components described in Section 2.3. Moreover, we keep metrics that reflect operational data in the system while discarding metrics that reflect the health of the software component (as we do not aim at detecting failures in the system). For example, in a Kafka cluster, metrics like `IsrShrinksPerSec`[1]`/IsrExpandsPerSec`[2] are important to track in case of expanding the number of Kafka brokers or removing partitions while the cluster is running. Monitoring these metrics checks if a replica is far behind the leader's offset, or if it has not contacted the leader for some time. However, this kind of metrics is excluded from our study as we do **not** change the deployment settings of the system while it is running, i.e., we do not increase the number of Kafka partitions or the replication degree while the system is running and thus monitoring such metrics is not essential. We also assume that deployed systems are healthy and that there is no failure in the system. For example, we do not consider a case where one Kafka node fails (dies/shuts down) and the partition's

[1] `IsrShrinksPerSec` refers to the fact that when a broker goes down, ISR (in-sync replicas) for some of the partitions will shrink. When that broker is up again, ISR will be updated once the replicas have fully caught up. Other than that, the expected value for both ISR shrink rate and expansion rate is 0.

[2] `IsrExpandsPerSec` refers to the fact that when a broker is brought up after a failure, it starts catching up by reading from the leader. Once it is caught up, it gets added back to the ISR.

leader on that node dies. Another example of such metrics is the `LeaderElectionRateAndTimeMs` metric that reports the rate of leader elections (per second) and the total time the cluster went without a leader (in milliseconds). In our case, we do not consider failures and thus we do include these kinds of metrics.

We define a set of prefixes described in Table 3.1 to refer to the software components in the studied cases. The prefix *CL* refers to metrics that are exported by the Kafka producer (client) which is an application that we use as a source of data in our case study [22]. The prefix *KSP* refers to metrics for which the values are computed based on values of Kafka metrics and Spark metrics i.e., the `KSP_consumer_lag` is computed based on the difference between a Kafka metric representing the offset of the last message that has been written into the messaging system and a Spark metric represents the offset of the last message that has been consumed.

As metrics are collected over several instances, partitions and/or replicas of a component, it should be mentioned that the values we study are aggregated over all instances of that component.

Component name	**Client**	**Zookeeper**	**Kafka**	**Spark**	**Cassandra**	**Kafka/Spark**
Prefix	CL	ZK	KF	SP	CS	KSP

Table 3.1 – The prefix of the software components names in the data processing pipeline.

As we have seen in Section 3.1, metrics of different kinds can be used to study the performance of distributed systems. We use a high-level classification to categorize metrics exported by all the components of the data processing pipeline. We classify metrics based on their purpose and the kind of information they provide. For example, we consider that a metric that shows the size of a queue of pending requests, belongs to the **Queue Size (QS)** type. Table 3.2 shows the representative types of the exported metrics.

Metric Type	**Description**
Resource Consumption (RC)	Represents the average percentage of resource utilization (i.e., CPU, memory, network, and disk IO).
Idle Threads (ITD)	Represents the average percentage of threads that are idle (i.e., threads that wait for an incoming request to process or wait for a reply from other threads).
Error Rate (ER)	Represents the average number of messages sent per second that resulted in errors. This type includes, for example, the error rate metric of a client that is issuing requests to Kafka brokers. The client error rate metric refers to the fact that the client is dropping messaging as the broker is overloaded.
Queue Waiting Time (QWT)	Represents the average request waiting time (in ms) in a queue before it has been served by the server. As an example, in the Kafka cluster case study, requests wait in a queue, after they have been received by the Network threads, to be served by other threads called the Handler threads.
Queue Size (QS)	Represents the size of a queue in terms of pending requests. This queue might be for incoming requests that have been received from a client but not yet served by the server. Another kind of queues might be for responses to client requests that have not been sent back to the requesting clients.
Latency (LY)	Represents the average amount of time that a system needs to process a message/request and send back a reply to the issuing client. More generally, latency means the time for any operation to complete. For example, latency in Kafka is defined by the time from when a message is published by the producer to when the message is read by the consumer.
Processing Time (PT)	Represents the average amount of time it takes to process a client request. This type includes, for example, the processing time that a stream processing engine (i.e., Spark Streaming) needs to process one batch of data within the streaming batch interval.
In/Out Data (IOD)	Represents the average number of requests/responses received/sent per time unit (i.e., second). For example, this type might include the number of requests that Kafka producer is able to send per second.
Uncategorized (UC)	Represents the set of metrics that do not fall in any of the previous categories. For example, the metrics that represent the number of client connections for a Zookeeper server are put in this category.

Table 3.2 – Representative types of exported metrics.

3.2.1 Resource consumption metrics

The resource consumption metrics are described in Table 3.3. These metrics represent the average utilization of CPU, memory, network, and disk IO.

Metric	Description
`cpu_usage`	The average percentage of CPU utilization.
`memory_usage`	The average percentage of memory utilization.
`network_sent`	The average percentage of network utilization for sent data.
`network_received`	The average percentage of network utilization for received data.
`disk_read`	The average percentage of disk utilization for read operations.
`disk_write`	The average percentage of disk utilization for write operations.

Table 3.3 – Metrics exported on the hardware level (resource consumption metrics).

3.2.2 Zookeeper performance metrics

As mentioned earlier, Zookeeper is a coordination service that is used by Kafka to store metadata about the Kafka cluster. Performance metrics exported by Zookeeper are described in Table 3.4.

Metric	Description	Type
`ZK_outstanding_requests`	The average number of queued requests in the Zookeeper server. Requests are queued if Zookeeper receives more requests than it can process.	Queue Size
`ZK_request_latency`	The average amount of time it takes for the Zookeeper server to respond to a client request.	Latency
`ZK_num_alive_connections`	The average number of client connections for a ZooKeeper server.	Uncategorized
`ZK_packets_received`	The average number of packets received.	In/Out Data
`ZK_packets_sent`	The average number of client packets sent including both responses and notifications.	In/Out Data

Table 3.4 – Metrics exported by Zookeeper.

The metrics exported in Table 3.4 reflect the way Zookeeper performs and interacts with the other components in the system. Zookeeper receives requests from clients (Kafka brokers, producers, and consumers). If the clients send requests faster than Zookeeper can process them, then Zookeeper queues the unprocessed requests and thus the `outstanding_requests` metric is an indicator of the loaded Zookeeper server. On the other hand, if a request waits for a while to be served, this affects the average request latency and thus the `avg_request_latency` is important to export. Furthermore, as both Kafka brokers and consumers communicate with ZooKeeper, this communication means that ZooKeeper could become a bottleneck, as ZooKeeper processes requests serially. Tracking the number of clients connected to it and the number of bytes sent and received over time could help diagnose performance issues. Thus metrics like `num_alive_connections`, `packets_received`, and `packets_sent` are exported.

3.2.3 Kafka producer (client) performance metrics

Kafka producers push messages to Kafka brokers (more precisely to the broker topics) for consumption. To ensure a steady stream of incoming messages sent by the producers, a set of metrics should be tracked. Table 3.5 describes these metrics which are exported by Kafka producers (clients).

Internally, the client buffers messages per partition before they are sent to the broker. After enough data has been accumulated or enough time has passed, the accumulated messages are removed from the buffer and sent to the broker. However, the messages wait longer in the buffer if the server is not able to process

the received messages as fast as they are sent. Thus monitoring the average time that the messages spend in the accumulator buffer reflects the internal interactions between the client and the server. On the other hand, a message results in an error if the client does not receive an acknowledgment from the server. The message error rate increases when the server is busy and it is unable to serve more messages. A high value of this metric refers to the fact that the producer (client) is dropping messages that it is trying to send to the Kafka brokers as these brokers are overloaded.

Furthermore, the request latency measures the amount of time between when a client sends a request until it receives an acknowledgment from a broker, which impacts the client performance (throughput), and thus is useful to track. A broker sends a response to the requesting client when the data has been received. *Received*, in this context, has different meanings depending on the acknowledgment setting:

- *acks = 0*: The client receives an ack when the partition leader has delivered the message.
- *acks = 1*: The client receives an ack when the leader has written the message to disk.
- *acks = -1* (or *all*): The client receives an ack when the leader has received confirmation from all replicas that the data has been written to disk.

When a client sends a request to a broker, the client marks this request as an incomplete request since its response has not been received yet from the broker. The incomplete requests wait in a buffer that is tracked by `request_in_flight` metric. Also, metrics like `request_rate` and `response_rate` reflect the traffic between the clients and the brokers and are very useful to monitor as they show if the brokers can sustain processing a rapid flow of data or not.

Metric	Description	Type
`CL_msg_queue`	The average time record batches spent in the message accumulator (in ms).	Queue Waiting Time
`CL_error_rate`	The average number of records sent per second that resulted in errors.	Error Rate
`CL_request_latency`	The average client request latency (in ms).	Latency
`CL_request_in_flight`	The average number of in-flight requests awaiting a response from the server.	Queue Size
`CL_request_rate`	The average number of requests sent per second.	In/Out Data
`CL_response_rate`	The average number of responses received per second.	In/Out Data

Table 3.5 – Metrics exported by Kafka producers (clients).

3.2.4 Kafka broker performance metrics

As mentioned earlier, the interactions between Kafka producers and Kafka brokers follow a client-server pattern. Within a server instance, there are two main pools of threads. First, the *Network threads* are responsible for handling new connections, requests from and responses to clients. These threads store the requests in a queue where they wait to be served. Second, the *Handler threads* are responsible for polling the client requests from the queue and processing them. Figure 3.1 shows an abstract overview of the client-server architecture in a Kafka cluster.

Based on the architecture that is depicted in Figure 3.1, a set of performance metrics that are exported by a Kafka broker must be tracked. Table 3.6 describes these performance metrics.

When a client sends a request to a Kafka broker, a Network thread receives this request and stores it in the request queue until it gets served by the Handler thread in the broker.

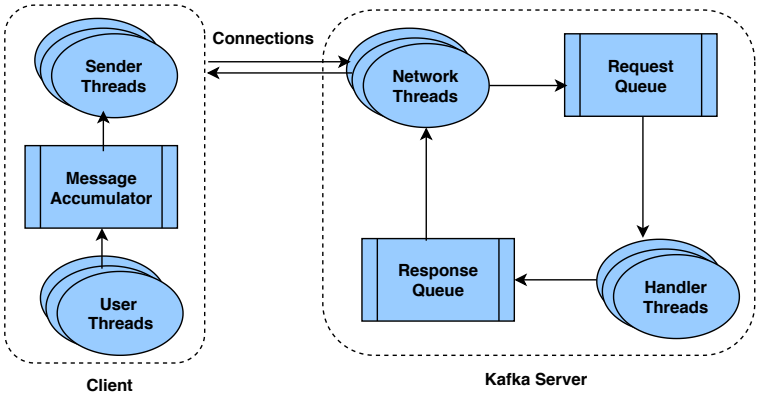


Figure 3.1 – An overview of the client-server architecture in a Kafka cluster [127].

The Handler thread in its turn processes the client request and puts the response in the response queue allowing the Network thread to send the response back to the client. Based on that, metrics like `network_threads_usage(%)`, `handler_threads_usage(%)`, `response_queue_time(ms)`, `req_queue_time(ms)`, `req_processing_time(ms)`, and `req_queue_size` are useful to track the status of clients requests.

Besides producer requests, the Kafka brokers have to serve other kinds of requests, i.e., the follower requests and the consumer requests. Followers (replicas) try to be in-sync with the leader by replicating messages. In-sync replicas constantly send *fetch* requests to the leader; these requests are the same as request that consumers send to consume produced messages. In these requests, they send the offset that they are waiting to receive next. Thus, metrics that monitor the time needed to fulfill followers and consumers requests are considered since they represent other kinds of operations that the brokers are responsible for. However, incomplete requests (requests that require a longer time to be considered complete) wait in special queues called *purgatory*. Monitoring these queues points out how many requests are being held by brokers, waiting to be satisfied. These values might be an indicator of how much the brokers are busy.

3.2.5 Spark performance metrics

Spark Streaming integrates with Kafka in a receiver-less approach (*direct* approach) [29]. Figure 3.2 illustrates at a high level the Kafka/Spark Streaming direct integration approach. In the direct approach, instead of using receivers to receive data, the Spark Driver periodically queries Kafka for the latest offsets in each topic/partition and accordingly defines the offset ranges to process in each batch. Furthermore, with `directStream`, Spark Streaming creates as many RDD partitions as there are Kafka partitions to consume, which all read the data from Kafka in parallel. So there is a one-to-one mapping between Kafka and RDD partitions. All performance metrics exported by Spark are described in Table 3.7.

When Spark receives data from Kafka for a given batch interval and creates the corresponding RDDs, an RDD processing is scheduled by the driver's `jobscheduler` as a job. At any given point in time, only one job is active. So, if one job is executing, the other jobs are queued. However, the time spent from when the collection of streaming jobs for a batch was submitted to when the first streaming job was started refers to the `Scheduling_delay`. After all the streaming jobs of a batch are started, the time spent to complete refers to the `processing_Time`.

On the other hand, tracking how much lag there is between the last message that has been written to Kafka and the last message that has been consumed by Spark Streaming gives an intuition of how fast

Metric	Description	Type
KF_network_threads_usage	The usage, in percent, of Network Handler threads. The Network Handler threads are responsible for reading and writing data to the clients across the network.	Idle Threads
KF_handler_threads_usage	The usage, in percent, of Request Handler threads. The Request Handler threads are responsible for serving the client requests, which includes reading or writing the messages to disk. As such, as the Kafka brokers get more heavily loaded, there is a significant impact on this thread pool.	Idle Threads
KF_response_queue_time	The response waiting time (in ms) in the response queue before it is sent to the client.	Queue Waiting Time
KF_request_queue_time	The request waiting time (in ms) in the queue before it is served by the server. This queue is between the network threads and the request handler threads.	Queue Waiting Time
KF_request_processing_time	The time (in ms) it takes to process a request at the partition leader.	Processing Time
KF_request_queue_size	The size of the requests queue (pending requests that are not served yet by the server).	Queue Size
KF_response_queue_size	The size of the responses queue (pending responses).	Queue Size
KF_fetchfollower_queue_time	The time (in ms) the fetch follower requests wait in the fetch request queue.	Queue Waiting Time
KF_fetchconsumer_queue_time	The time (in ms) the fetch consumer requests wait in the request queue.	Queue Waiting Time
KF_fetchfollower_total_time	The total time (in ms) to serve a fetch follower request.	Processing Time
KF_fetchconsumer_total_time	The total time (in ms) to serve a fetch consumer request.	Processing Time
KF_producer_purgatorySize	The size of the purgatory queue where requests, that need a long time to be processed, are stored. There is a noticeable increase in the size of this queue when the acknowledgment option is set to all. The producer purgatory holds client requests that have not yet met their criteria to succeed but also have not yet resulted in an error.	Queue Size
KF_fetchfollower_purgatorySize	The number of requests waiting in the fetch purgatory. The fetch purgatory holds fetch requests that have not yet met their criteria to succeed but also have not yet resulted in an error.	Queue Size
KF_fetchfollower_throttletime	The average throttle time (in ms). The broker may delay fetch requests in order to throttle a follower.	Queue Waiting Time
KF_producer_throttletime	The average throttle time (in ms). The broker may delay clients requests in order to decrease the load on the server side.	Queue Waiting Time

Table 3.6 – Metrics exported by Kafka servers (brokers).

the consuming component in the system is compared with the publishing one. This is measured by the KSP_consumer_lag metric.

Metric	Description	Type
SP_scheduling_delay	The time (in sec) a batch waits in a queue for the processing of previous batches to finish.	Queue Waiting Time
SP_processing_time	The time (in sec) spent to complete all the streaming jobs of a batch.	Processing Time
SP_total_delay	The time (in sec) spent from submitting to complete all jobs of a batch.	Latency
SP_number_records	The number of processed streaming records.	In/Out Data
KSP_consumer_lag	How much lag (in msg) there is between the last message that has been written into the messaging system and the last message that has been consumed.	Queue Size

Table 3.7 – Metrics exported by Spark Streaming.

3.2.6 Cassandra performance metrics

In our data processing pipeline, after a message has been consumed and processed by Spark, the results are stored in Cassandra. Spark connects to Cassandra cluster through Spark Cassandra Connector API [40]. This connector allows exposing Cassandra tables as Spark RDDs, writing Spark RDDs to Cassandra tables, and executing arbitrary Cassandra query language (CQL) queries in Spark applications. Figure 3.3 illustrates Spark/Cassandra integration at a high level. Performance metrics exported by Cassandra are described in Table 3.8.

When results are sent to Cassandra, a set of metrics can be tracked to identify slowdowns, hiccups, or pressing resource limitations in the cluster. Monitoring both write_request_latency and read_request_latency gives a better understanding of how much the cluster is being busy and whether the workload tends to be read-heavy or write-heavy. On the other hand, when a Cassandra node receives

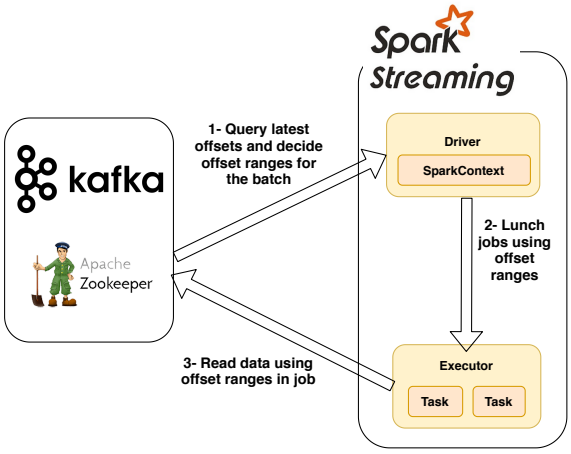


Figure 3.2 – High level description of Kafka/Spark Streaming direct integration [34].

requests faster than it can process, these requests wait in a queue to be served. Monitoring the number of waiting requests reflects the activities of the systems and which components are faster than the other.

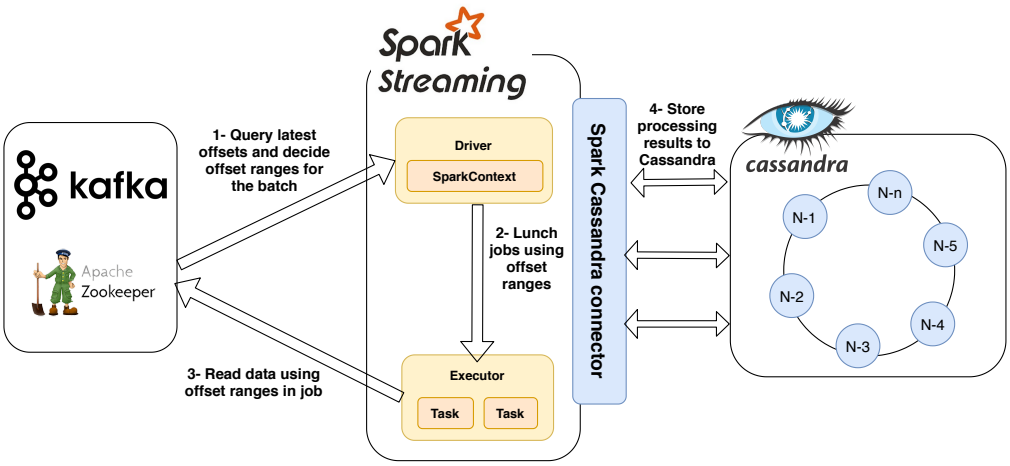


Figure 3.3 – High level description of Kafka/Spark/Cassandra integration.

In this section, we have presented many performance metrics that can be exported by each software component of the data processing pipeline. The question to answer now is: *Is it possible to identify a set of key metrics that can be used as reliable and general indicators of performance bottlenecks in a given multi-tier distributed system?* To reply to this question, we apply our proposed analytical study as explained in the next section.

3.3 The proposed methodology to identify reliable indicators of performance bottlenecks

In this section, we describe the proposed methodology of the analytical study to identify reliable indicators of performance bottlenecks in the data processing pipeline. First, we present the assumptions we consider to apply our methodology. We present the research questions that the study is aiming to answer. Then, we describe cases that the study covers. We present the definition of *global throughput* and bottleneck.

Metric	Description	Type
CS_write_request_latency	The write response time (in ms).	Latency
CS_read_request_latency	The read response time (in ms).	Latency
CS_request_timeout	The number of read/write requests not acknowledged within a configurable timeout window. Request timeout exception reflects the incomplete (but not failed) handling of a request. Timeouts occur when the coordinator node sends a request to a replica and does not receive a response within the configurable timeout window. Timeouts are not necessarily fatal (the coordinator will store the update and attempt to apply it later) but they can indicate network issues or even disks nearing capacity.	Uncategorized
CS_pending_tasks	The total number compaction tasks in queue.	Queue Size

Table 3.8 – Metrics exported by Cassandra.

Finally, we describe how to analyze the evolution of the exported metrics to identify the reliable indicators of performance bottlenecks.

3.3.1 Assumptions

In this thesis, we make the following assumptions:

- We consider *Global Throughput* as the main performance metric that we are interested in (see below the definition of the *Global Throughput*).
- We are not interested in satisfying some objective functions (service level objectives SLOs) (i.e., minimizing the request-response time [109, 121, 162]). Instead, we aim at improving the *Global Throughput* of the data processing pipeline without predefining the throughput value that we should achieve.
- We assume that the observed system has a steady-state behavior and that the performance-limiting component is stable over time, as this corresponds to the case where we only inject a homogeneous load. Simply put, we do not consider setups exhibiting transient or rapidly alternating bottlenecks [148, 149].
- We assume that the running system is well tunned in terms of software settings, e.g., the selected option for acknowledgment management in Kafka can not be changed. In other words, the system throughput can be improved mainly by provisioning more hardware resources. For instance, in a Kafka cluster use case, adding more server machines can lead to an improvement in the *Global Throughput* of the system.
- We do not consider the deployment of systems where the hardware resources can be shared between the system software components. This means that the system components are not collocated on the same machines, instead, each component has its dedicated machines for deploying its instances.

3.3.2 The methodology

In this section, we present our proposed methodology for the analytical study and how to apply it. We first, list the questions the study is trying to address. We explain the different use cases the study is considering. We define *Global Throughput* as the main performance metric that we are interested in. We also explain how we define performance bottlenecks in the studied cases and how we identify them. Finally, we discuss and summarize our approach.

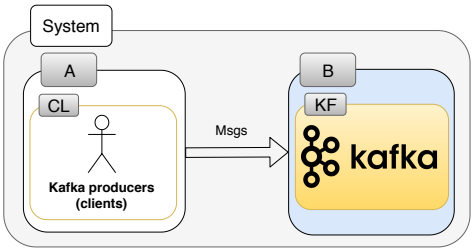


Figure 3.4 – Illustration of a system comprising a pair of interacting components A and B.

3.3.2.1 Research questions

Throughout this analytical and empirical study, we strive to answer the following questions:

- **Question 1**: Are there metrics (exported by the data processing pipeline) that we can rely on to identify performance bottlenecks in the system? And if these metrics do exist, are they specific to some setups or are they *generally* good enough to be able to always rely on them in identifying performance bottlenecks?
- **Question 2**: In the case of the existence of the reliable indicators of performance bottlenecks, what are these metrics? How can we identify them? What are their characteristics, i.e., how do these metrics behave?
- **Question 3**: How fine-grained are the conclusions that we can draw from these metrics, i.e., can we just conclude if the bottleneck is on the client side or the server side or can we also conclude in the case where the server is composed of multiple components (e.g., the server is composed of Kafka, Spark, and Cassandra) about which component is limiting the performance?

3.3.2.2 Studied cases

Our study covers different use cases and configurations of the data processing pipeline. For each configuration, we make the distinction between (i) the client side, and (ii) the server side. The client side represents the system component that is responsible for injecting the data into the system. The server side represents the system component that encompasses all other system components that participate in processing the data injected in the system. The server side can include one or multiple components.

For each application, we study the performance of a pair of interacting components A and B where A represents the client side and B represents the server side (e.g., Kafka clients issuing requests to Kafka servers). Figure 3.4 illustrates a system comprising a pair of two interacting components A and B where the component B includes a single component(Kafka). Figure 3.5 illustrates a system comprising a pair of two interacting components A and B where the component B is composed of two software components: Kafka and Spark Streaming. Figure 3.6 illustrates a system comprising a pair of two interacting components A and B where the component B is composed of three software components: Kafka, Spark Streaming, and Cassandra.

Note that, like in the example mentioned just above, a given component A or B can be implemented as several instances (e.g., multiple shards or replicas); unless mentioned otherwise, we consider the notion of a *logical* component, which regroups all the corresponding instances. We vary the input load injected by the component A and measure the impact on the throughput of the component B.

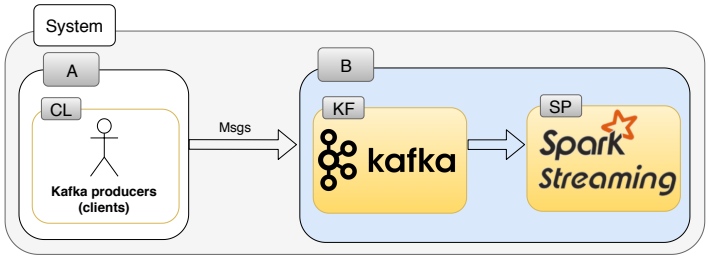


Figure 3.5 – Illustration of a system comprising a pair of interacting components A and B, where B is composed of Kafka and Spark Streaming.

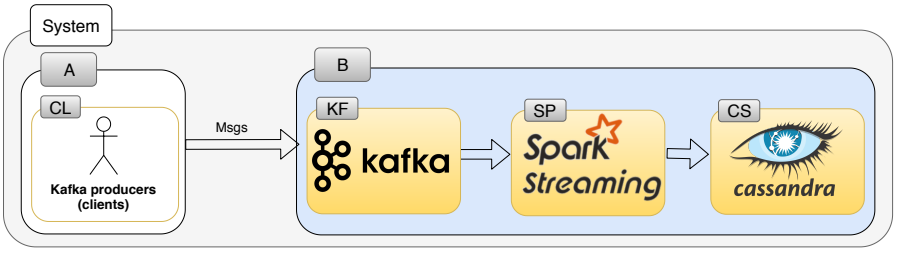


Figure 3.6 – Illustration of a system comprising a pair of interacting components A and B, where B is composed of Kafka, Spark Streaming, and Cassandra.

3.3.2.3 Global throughput definition

The main performance metric we are interested in is the *global throughput* of the system, i.e., the amount of work (or tasks) that the system can handle per time unit.

The *global throughput* in the case of a full data processing pipeline is the number of tasks/messages that the system processes per time unit. Processing a message includes the following steps: (i) a client sends a message to the messaging system Kafka, (ii) Spark streaming pulls the message to process it (e.g., applying Wordcount), and (iii) Spark streaming stores the results (e.g., the frequency of the incoming words) into the database, Cassandra.

In the case of a Kafka cluster where a Kafka client is issuing requests to the Kafka broker, the *global throughput* of the system is the number of messages that are sent by the client and that are written to Kafka commit log per time unit.

3.3.2.4 Definition of the notion of bottleneck

To identify the limiting component in the system, we strive to identify the point of *peak* throughput, where, while increasing the number of clients, the component limiting the performance of the system is shifting from one side to the other. Before this point, we consider that component A is the *bottleneck* component, meaning that performance could be improved by increasing the intensity of the load injected by A. Conversely, at this point *and beyond*, we consider that component B is the *bottleneck* component, meaning that performance can only be improved by provisioning more hardware resources for B (or by a better tuning of B's settings). Note that, there might be a *gray zone* where it is ambiguous to identify if the client (component A) is still the bottleneck or if the bottleneck shifts completely to the server side (component B).

We say that B (i.e., the server) is a bottleneck when B reaches its maximum capacity in terms of *global throughput* with the given hardware resources. Note that we use the notion of A (i.e., the client side) being a bottleneck to refer to the fact that the server has not reached its maximum capacity with the current load injection.

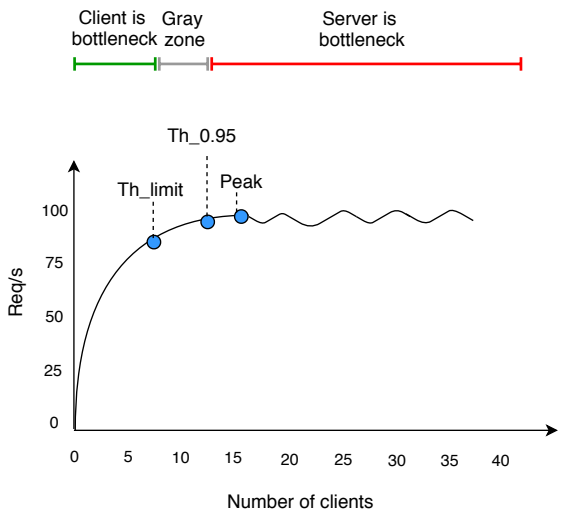


Figure 3.7 – Illustration of bottleneck identification based on throughput measurement.

3.3.2.5 Identification of the bottleneck

To identify the bottleneck component in the system, we consider an experiment where we measure the evolution of the *global throughput* of the system while increasing the number of clients. To analyze the obtained results, we use the following approach. First, we consider a margin of 5% for the measured values near the peak: for a given setup, if a measured throughput value is within 5% of the observed peak throughput value, we consider that we reach the maximum capacity of the server. Hence, for each experiment, we indicate the *peak throughput range* that we observe: $[peak \times 0.95; \quad peak]$.

Second, we define an additional *safety margin* by considering another threshold M to split the data points of a given experiment. We compute the throughput Th_{limit} as follows: $Th_{limit} = Th_{0.95} \times (1 - M)$ where $Th_{0.95} = peak \times 0.95$. We explain how we find the value of M in the experimental evaluation in Section 3.4. We consider the points where the system throughput is less than Th_{limit}, the points where the client is the bottleneck. We consider the points where the system throughput is greater than $Th_{0.95}$, the points where the server is the bottleneck. The gray zone is the set of points where the throughput belongs to the range $[Th_{limit}; \quad Th_{0.95}]$. These points represent the case where it becomes difficult to distinguish which side of the system is the bottleneck i.e., both improving A and B can improve the global throughput of the system. In other words, in the gray zone, tuning both the client and the server by provisioning more hardware resources (i.e., adding more nodes) leads to an improvement in the global throughput of the system.

Figure 3.7 shows an example of system throughput measured in requests per second as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the throughput. The figure illustrates the bottleneck identification of a setup with a peak throughput of 100 req/s and a $M = 10\%$, we obtain $Th_{0.95} = 95$ req/s, and $Th_{limit} = 85.5$ req/s.

3.3.2.6 The analysis of the metrics

In this section, we strive to answer the research question concerning identifying the characteristics of metrics that act as *reliable* indicators of performance bottlenecks. Throughout our analysis, we are exporting a large number of metrics at the hardware and the software levels. The first question that we need to answer is what kinds of patterns in the evolution of metrics can be identified as a reliable pattern, i.e., a pattern that allows identifying a bottleneck when it shifts from one component to another. In this section, we present at

the abstract level all the patterns that we have observed during our study and we explain why we categorize them into 4 main categories.

We insist on the fact that trend changes in metrics graphs are of high importance. By trend, we refer to the evolution of metric values, e.g., it is constant or it is increasing. By trend change, we mean that there is a change in the evolution, e.g., it was constant and it starts increasing. Our study relies on associating changes in trends with changes in the side of the bottleneck.

We identify 4 representative categories of patterns concerning the relation between the change in trends and values of a metric and the identification of the bottleneck. These categories are: (i) Reliable patterns where we can easily define a threshold that differentiates a setup where A or B is the bottleneck; (ii) Partially reliable patterns where a single value is not enough to identify the limiting component, but two consecutive values when varying the number of clients, that define the trend, are enough; (iii) Not reliable patterns where even with the value and the trend it is not possible to identify the bottleneck component; (iv) Misleading patterns where a metric looks reliable or partially reliable but the change in its behavior does not occur when the bottleneck shifts from A to B. These categories are identified based on observing the behavior of the metrics while the system is running and the number of the clients is increasing (until the system reaches its saturation i.e., maximum capacity). We explain in detail each of these categories below.

- **Category-1– Reliable patterns**: This category includes the patterns where metrics behavior significantly changes at the point where the bottleneck shifts from the client side to the server side. We consider the metrics that fall in this category as **reliable** indicators of performance bottlenecks for a given configuration as a value of these metrics is enough to decide if the bottleneck is on the client side or the server side. Figure 3.8 illustrates the set of patterns that represents the evolution of *reliable* metrics. Each sub-figure shows the evolution of the values of a metric as a function of the number of clients. The X-axis represents the number of the clients and the Y-axis represents the metric values. The *green* points correspond to the metrics values at the point where the bottleneck is on the client side. In other words, these points correspond to the phase where the system throughput is less than Th_{limit}. The *gray* points correspond to the metrics values at the point where it becomes difficult to accurately identify where the bottleneck is (client or server). In other words, these points correspond to the phase where the system throughput is equal to or greater than Th_{limit} and less than $Th_{0.95}$. Finally, The *red* points correspond to the metrics values at the point where the bottleneck is on the server side. These points correspond to the phase where the system throughput is equal to or greater than $Th_{0.95}$ (i.e., the system has reached its saturation point and an increase in the injected load will not improve the global throughput of the system).

  All the patterns presented in Figure 3.8 are reliable because a significant change in the values of the metric occurs when the server reaches its maximum capacity (red points) or in the gray zone. We can further categorize these patterns as follows:

  Constant-linear-constant Figures 3.8(a), (b), and (d) show a significant change in the metrics values at the point where the bottleneck shifts from the client side to the server side. For both patterns in Figures 3.8(a) and (b), the values evolve constantly at a given value V where the bottleneck is the client side and then the behavior changes by showing an increase in the metric values from the value V to a new value W, then again the behavior remains constant at the new value W. In Figure 3.8(d), the metrics values remain constant at a given value V when the bottleneck is on the

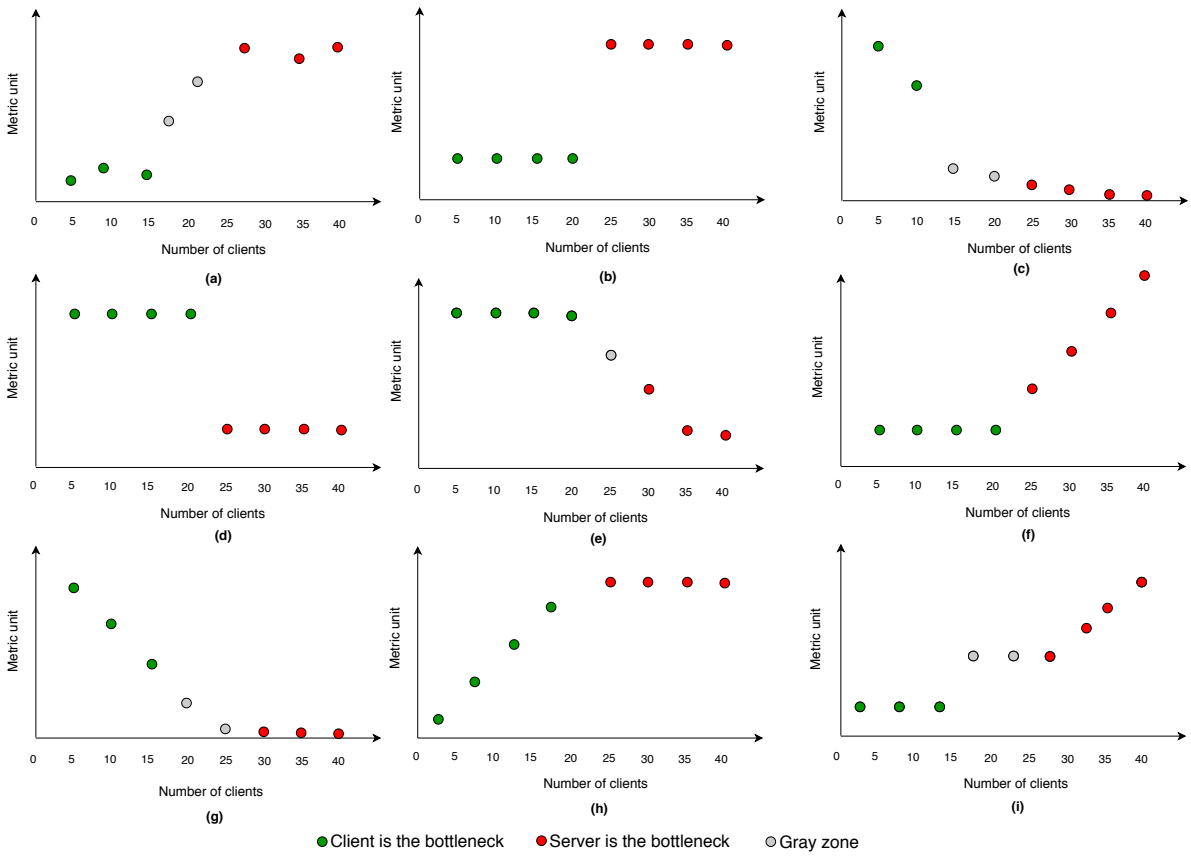


Figure 3.8 – Illustration of *reliable* patterns (category-1).

client side and when the bottleneck shifts to the server side, the metrics values drop to another value W and remain constant at that value.

Constant-linear Figures 3.8(e) and (f) show that the metrics values remain constant at a given value while the bottleneck is on the client side and the value increases/decreases linearly when the bottleneck shifts to the server side. In both patterns, the significant change in both the metrics values and their behaviors at the point where the bottleneck shifts from one side to the other allows us to determine at any given point of the metrics values, where the bottleneck lies.

Linear-constant Similarly to Figures 3.8(e) and (f) but in a reverse way, Figures 3.8(c), (g), and (h) show that the metrics values increase/decrease linearly while the bottleneck is on the client side and the behavior remains constant at the point where the bottleneck shifts to the server side. The significant change in both the metrics values and their behaviors at the point where the bottleneck shifts from one side to the other allows us to determine at any given point of the metrics values, where the bottleneck lies.

Constant-constant-Linear Figure 3.8(i) shows two constant behaviors of the metrics values at two different given values V and W. Then, these values increase at the point where the bottleneck shifts to the server side. Metrics that follow this kind of pattern are considered as *reliable* indicators of performance bottlenecks as at any given point of the metrics values, we can easily identify the limiting component in the system.

- **Category-2– Partially reliable patterns**: This category includes the patterns where giving a single value of the metric does not allow concluding about the limiting component in the system. However,

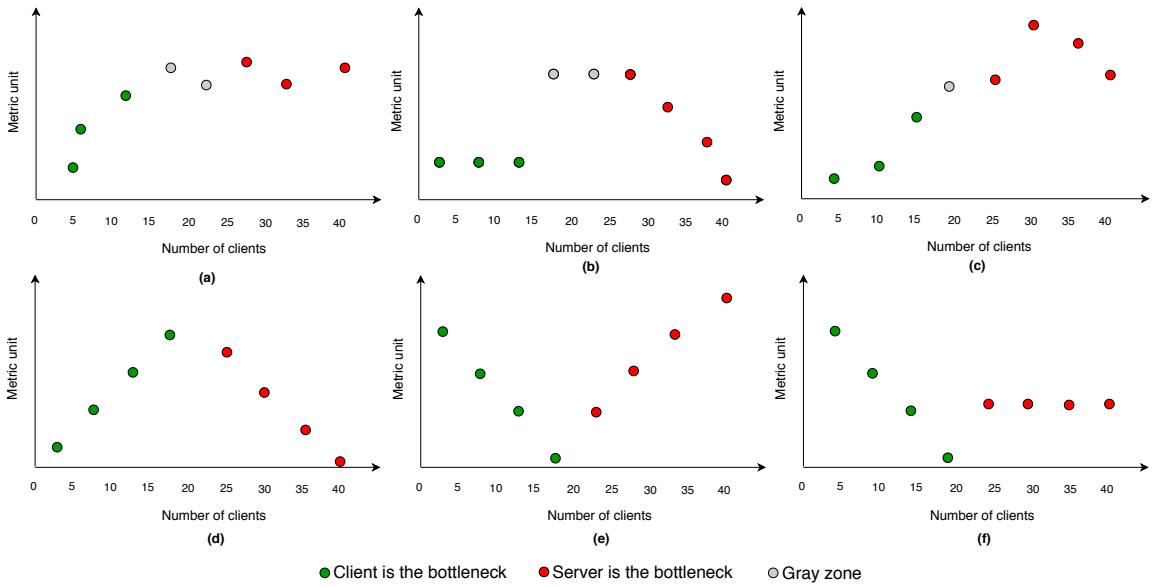


Figure 3.9 – Illustration of *partially-reliable* patterns (category-2).

having the trend of the evolution of the values (by giving two consecutive points) allows identifying the bottleneck component in the system. We consider metrics that fall in this category as **partially reliable** indicators of performance bottlenecks as they are more difficult to analyze.

Figure 3.9 illustrates the set of patterns that represents the evolution of *partially reliable* metrics. Each sub-figure shows the evolution of the values of a metric as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values.

In this category, we observe the following patterns:

Linear-periodic variation Figure 3.9(a) shows that the metrics values increase while increasing the number of clients. Then the metrics get spikes (up/down) in their values at the point where the bottleneck shifts from the client side to the server side. Having one value of the metric evolution does not allow us to identify on which side the bottleneck is. For example, if we are given one of the green points or one of the red points as shown in Figure 3.9(a), we will not be able to identify on which side the bottleneck is. On the other hand, if we are given two consecutive points (two greens or two reds), we will be able to know the trend of the points and thus identify the bottleneck side.

Constant-constant-linear Figure 3.9(b) shows a constant behavior of the metrics values at a given value V, followed by another constant behavior at another given value W, then a decrease in the values at the point where the bottleneck shifts to the server side. In this pattern, only the metric's trend that shows that the metrics values are decreasing when increasing the number of clients is a good indicator of shifting the bottleneck from the client side to the server side.

Linear-linear Figures 3.9(c), (d), and (e) show a clear change in the trend of the metrics values at the moment where the bottleneck shifts from the client side to the server side. However, the same metric value can be observed before and after the server reaches its maximum capacity. Thus metrics that follow this kind of pattern are considered as partially reliable indicators.

- **Category-3– Not reliable patterns**: This category includes the patterns where the values of the metrics does not vary wherever the bottleneck is or it varies in a linear way wherever the bottleneck is.

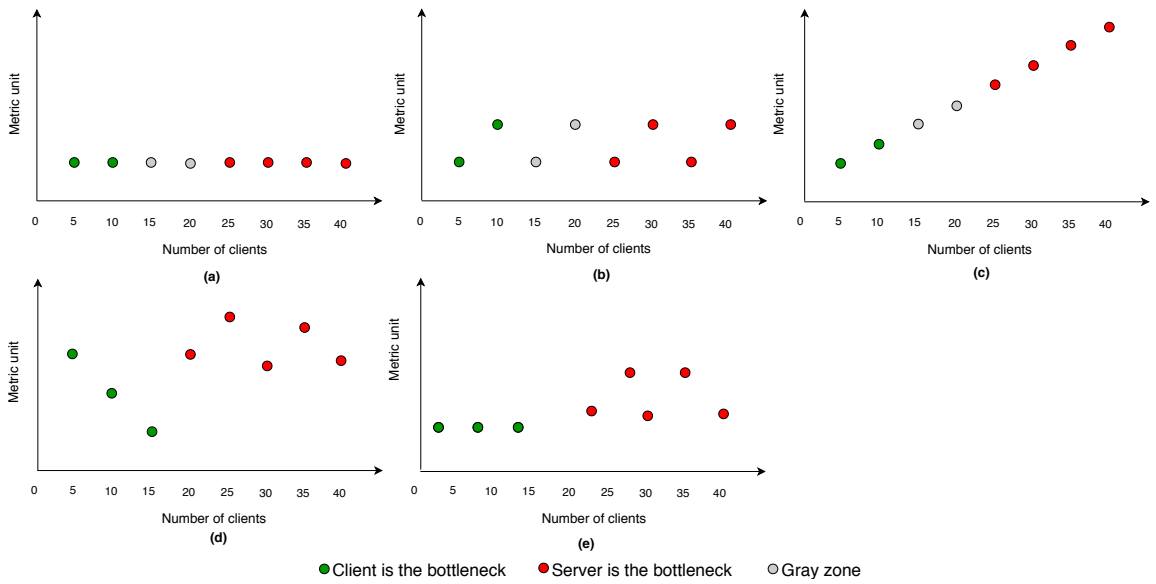


Figure 3.10 – Illustration of *not-reliable* patterns (category-3).

We consider the metrics that fall in this category as **not reliable** indicators of performance bottlenecks as their behavior does not change whether the bottleneck is on the client side or the server side. Figure 3.10 illustrates the set of patterns that represents the evolution of *not reliable* metrics.

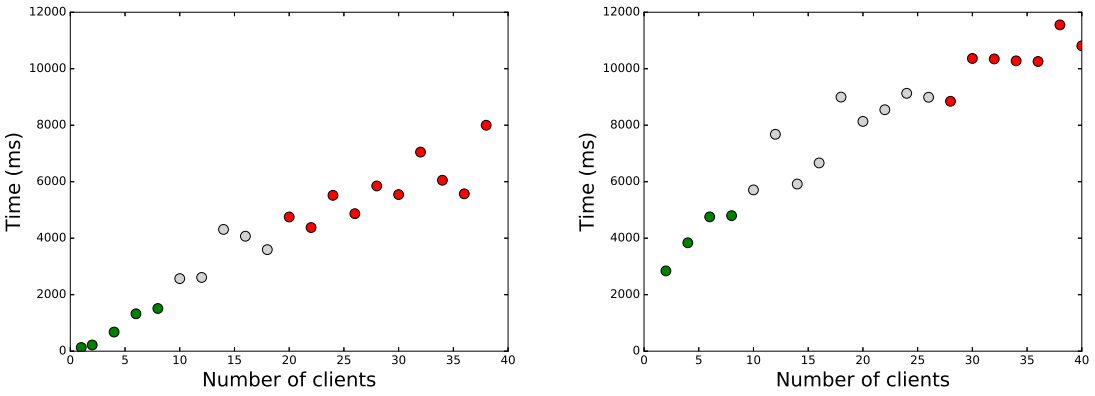
In this category we observe the following patterns:

Constant Figure 3.10(a) shows a steady evolution in metrics values during the whole experiments time and wherever the bottleneck is. In other words, metrics that follow this kind of pattern do not change their behavior regardless of the bottleneck location.

Periodic variation Figure 3.10(b) shows spikes (up and down) in the metric values in a regular way. Thus metrics that follow this kind of pattern cannot be indicators of performance bottleneck as they behave in the same way whether the bottleneck is on the client side or in the server side.

Linear Figure 3.10(c) shows a positive linear evolution of metrics values while increasing the number of clients. As the metrics have a constant evolution of their values without any change in their behavior, then it is difficult to identify on what side the bottleneck is or when the bottleneck shifts. We could set a threshold to identify when the bottleneck shifts from the client side to the server side. But setting a threshold value can be a difficult task, i.e., this threshold might require tuning when changing the system setup. For example, we can consider a Kafka cluster with two setups: `setup-0` and `setup-3` described in Table 3.10. The difference between the two setups is the size of the message: 100 and 10000 Bytes respectively. For both setups, we monitor `CL_msg_queue` metric. Figure 3.11 shows the evolution of the `CL_msg_queue` metric as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. For both setups, the `CL_msg_queue` metric evolves linearly. However, Figure 3.11 (a), shows that the bottleneck shifts to the server side at 22 clients and that the corresponding `CL_msg_queue` metric value is 4377 ms. Figure 3.11 (b), shows that the bottleneck shifts to the server side at 28 clients and that the corresponding `CL_msg_queue` metric value is 9016 ms. Consequently, we do not consider metrics that follow this kind of pattern as *reliable* indicators as it requires setting a threshold specific to a configuration to identify when the bottleneck shifts from the client side to

the server side.



(a) Avg. msg. waiting time(ms) for setup-0.

(b) Avg. msg. waiting time(ms) for setup-3.

Figure 3.11 – Evolution of `CL_msg_queue` metric for setup-0 and setup-3 of a Kafka cluster.

Change in behavior but not in values Figures 3.10(d) and (e) show metrics that evolve in a constant way when the bottleneck is on the client side. The metrics values decrease by a constant given value V or remain constant at a given value V. Then the behavior shows spikes (up/down) where the bottom values of the spikes are at the same value V as before the metrics change their behavior to a spiky pattern. Thus, observing the behavior of this kind of metric does not help in identifying the side where the bottleneck lies.

- **Category-4– Misleading patterns**: This category includes the patterns where the metrics look reliable or partially reliable. However, the change in the behavior of these metrics does not occur at the moment where the bottleneck shifts from the client side to the server side. We consider metrics that follow this kind of pattern as **misleading** indicators of performance bottlenecks as relying on these metrics might lead to wrong decisions, i.e., they might lead to a false identification of a bottleneck in the server side. Relying on such metrics, we might decide to provision more hardware resources to the server and pay unnecessary extra costs. Figure 3.12 shows a set of patterns where metrics change their behavior at a point that does not correspond to the right point where the bottleneck shifts from the client side to the server side.

3.3.2.7 Selecting indicators of performance bottleneck

Now, what we have defined the categories of patterns that metrics might follow, the question is how we select the indicators of performance bottlenecks. To answer this question, we define a set of rules as follows.

- If some of the exported metrics fall in **category-1**, then we conclude that there is a set of reliable metrics that we can rely on as indicators of performance bottlenecks in the given configuration of the data processing pipeline. If there is a reliable metric of performance bottlenecks, then we can identify the limiting component in the system based on the current observation of the metrics status.

- If at best, some of the exported metrics fall in **category-2**, we conclude that there is a set of partially reliable metrics that we can rely on as indicators of performance bottlenecks in a given configuration of the data processing pipeline. To rely on these metrics to identify the bottleneck, an observation of the metrics over a duration of time is required to get the trend of their behavior.

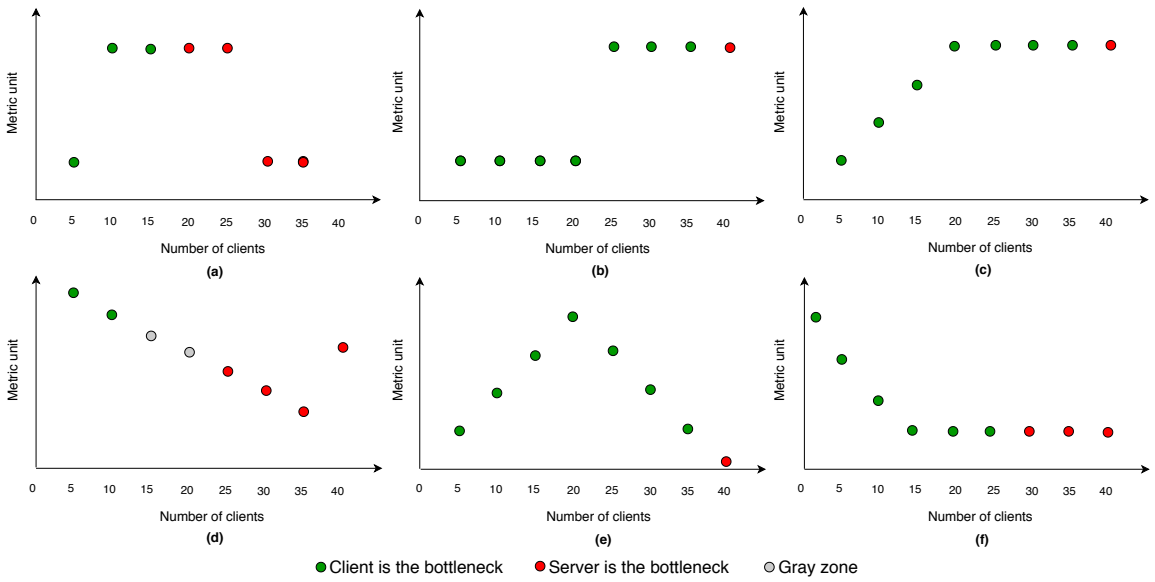


Figure 3.12 – Illustration of *misleading* patterns (category-4).

- Note that if a given metric behaves as a *reliable* or a *partially reliable* indicator for some setups and it behaves as a *misleading* indicator for other setups, then we consider this metric as a one that we should not rely on.
- If all exported metrics fall in **category-3**, we conclude that there are no reliable indicators of performance bottlenecks in a given configuration of the data processing pipeline.
- Finally, if the exported metrics fall in **category-4**, we cannot rely on these metrics as they might lead to a false identification of the performance bottlenecks.

3.3.2.8 Putting all the pieces together

In the previous sections, we have seen in detail the research questions the study tries to answer, the covered studied cases, the pattern categories of metrics evolutions, and how to select performance bottleneck indicators.

Now, we summarize the main steps to be run for the selection of these performance bottleneck indicators. Note that these steps are not specific to the data processing pipeline: they can be applied to other systems as we show for the Web stack use case in Chapter 5. Figure 3.13 summarizes the main steps of the analytical study as following:

- First, we define the *peak throughput* of the system. As mentioned before, the *peak throughput* represents the point where increasing the number of clients does not lead to any improvement in the *global throughput* of the system.
- Then, we define a safety margin equals to 0.95 to compute the *peak throughput range* which is: $[peak \times 0.95; \quad peak]$. This margin allows us to deal with variations in the measurements.
- We define a gray zone, where adding resources on client or on server sides leads to an increase in the throughput. The gray zone is in the range $[Th_{limit}; \quad Th_{0.95}[$ where $Th_{limit} = Th_{0.95} \times (1 - M)$ and $Th_{0.95} = peak \times 0.95$. Sections 3.5.1, 3.6.1, and 3.7.1 illustrate how to set the value of M for Kafka, Kafka/Spark Streaming, and full pipeline cases respectively.

- Based on the values of $Th_{0.95}$ and Th_{limit}, we analyze the evolution of the values of the exported metrics. We follow the methodology described earlier to analyze metrics and identify the pattern categories that they belong to.
- Finally, we apply the declarative set of rules that we defined for selecting useful indicators of performance bottleneck.

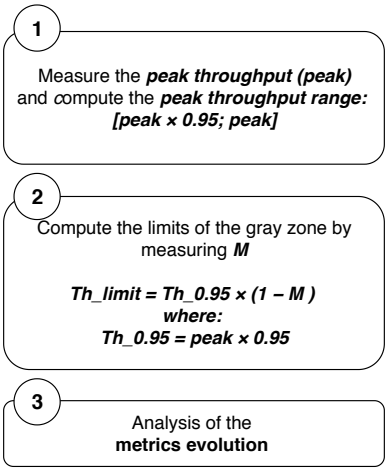


Figure 3.13 – Main steps of the analytical study for one given setup.

Figure 3.14, shows the main steps of selecting useful indicators of performance bottlenecks in a multi-tier distributed system. Note that running all these steps manually can be tedious and error-prone. Hence, in Chapter 4, we study how machine learning techniques can help us in this task.

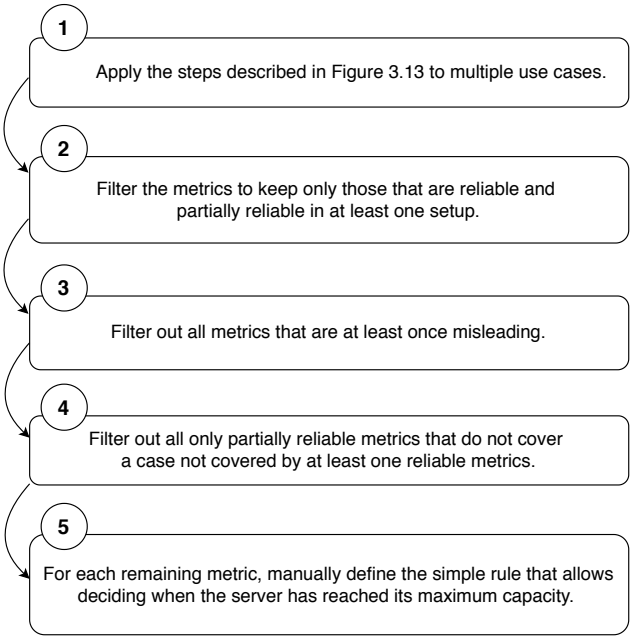


Figure 3.14 – Main steps of selecting useful indicators of performance bottlenecks.

In the rest of this chapter, we are going to apply the proposed analysis to multiple configurations of the data processing pipeline (Kafka alone, Kafka/Spark, Kafka/Spark/Cassandra), considering multiple setups

for each configuration of the data processing pipeline. Several questions remain open and will be answered through our experimental evaluation.

The first question relates to the application of our proposed methodology on real data. One may wonder whether it is possible to find a single value of M that can be applied to different setups and that can accurately identify the beginning of the gray zone. If a different value of M should be selected for each tested case, it would imply that each setup should be analyzed independently, making the analysis of a large number of setups and configurations impracticable. We show in the following sections, that for a given configuration of the data processing pipeline, it is possible to select a single value of M that can identify the beginning of the gray zone with good accuracy in different setups.

The other questions relate to the main objective of this chapter, that is, understanding the characteristics of metrics that can be a good indicator of performance bottlenecks in the data processing pipeline. The main questions that we aim at answering are:

- For each studied setup, are there metrics that act as reliable indicators of performance bottleneck?
- In the case of the existence of reliable metrics, do they always act as reliable indicators? Or does their behavior depend on the examined setup?
- Are resource consumption metrics sufficient as indicators of performance bottlenecks? More generally, can metrics from a single category defined in Table 3.2 be enough to identify performance bottlenecks?
- Can we, at least, find metrics that we can rely on, that is, metrics that are sometimes reliable, sometimes partially reliable, but that are never misleading?

We aim at answering these questions considering: i) Different configurations of the data processing pipeline; ii) Different setups for each configuration; iii) Different processing applications for the pipelines that include Spark Streaming.

3.4 Experimental evaluation

In this section, we apply the proposed methodology described in Section 3.3 on the data processing pipeline.

We examine three use cases of the data processing pipeline. The first use case is comprising only Kafka which receives messages from clients and writes them to the disk (Section 3.5). In the second use case, we connect the Spark Streaming component to the Kafka component. In this subsystem, Spark Streaming subscribes to Kafka to get messages, processes them, then writes the results to a file (Section 3.6). In the third use case, we connect a Cassandra cluster to the Kafka/Spark cluster. In this last case, results are stored in Cassandra instead of a file (Section 3.7).

Through this section, we first present the methodology of our evaluation including the hardware setup, the software setup, and the data (i.e., metrics) collection tools. Then, we present the different examined applications for the data processing pipeline.

3.4.1 Testbed and data collection

3.4.1.1 Hardware setup

We deploy the data processing pipeline on a fixed number of homogeneous machines (hereafter named *nodes*). We use 9 nodes, 3 for each software component: 3 nodes for the Kafka cluster, 3 nodes for the Spark Streaming

cluster, and 3 nodes for the Cassandra cluster. As mentioned earlier, Zookeeper instances are deployed on the same nodes as Kafka instances.

We do not change the deployment settings while the system is running. For example, we do not increase the number of Kafka partitions while increasing the number of clients.

The experiments are performed on the Grid'5000 testbed [1] using different sites: Lyon and Nantes. Table 3.9 shows the hardware configuration for all used clusters.

We use two sites as we wanted to examine different hardware configurations for deploying the server instances. We examine a setup where the server nodes have good processing power by having 2 8-core CPUs (Nova cluster). We also, examine a setup where server nodes have only a 1-core CPU (Sagittaire cluster). Finally, we examine a setup with different hardware configuration where we have SSD disks (Ecotype cluster). More details about these clusters are described in Table 3.9.

In the Lyon site, we use the Nova cluster as server nodes, where each host has the same hardware configuration with 2 8-core CPUs, 64GB of RAM, a 600GB HDD, and a 10 Gbps Ethernet interface. We use the other clusters (i.e., Orion, Hercule, and Taurus) as client nodes.

For some experiments in the Lyon site, we use the Sagittaire cluster where each host has the same hardware configuration with a 1-core CPU, 2GB of RAM, a 73GB HDD, and a 1 Gbps Ethernet interface.

In Nantes site, we perform experiments that use SSD disks, using Ecotype cluster where each host has the same hardware configuration with 2 10-core CPUs, 128GB of RAM, a 400GB SSD, and a 10 Gbps Ethernet interface.

Site	Cluster	Hardware configuration
Lyon	Nova	2 x 8-core CPUs, 64GB RAM, a 600GB HDD, and a 10 Gbps Ethernet interface
Lyon	Sagittaire	1-core CPU, 2GB RAM, a 73GB HDD, and a 1 Gbps Ethernet interface
Nantes	Ecotype	2 x 10-core CPUs, 128GB RAM, a 400GB SSD, and a 2 x 10 Gbps Ethernet interface.

Table 3.9 – Grid'5000 testbed hardware configuration [1].

3.4.1.2 Software setup

All experiments use Debian 8 with a 3.16.0 Linux kernel, OpenJDK version 1.8.0_131, Scala 2.11, ZooKeeper 3.4.10, Kafka 0.11, Spark Streaming 2.1.0, and Cassandra 3.0.9.

3.4.1.3 Data collection

The data related to hardware resource consumption (i.e., CPU, memory, and IO) are obtained via `psutil` (python system and process utilities)[3]. It is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, and network) in Python.

The data on the software component level are obtained via Java Management Extensions (JMX) [54].

We apply the analytical study on 35 different setups of the data processing pipeline (15 setups for the Kafka cluster use case, 10 setups for the Kafka/Spark Streaming use case, and 10 setups for the Kafka/Spark Streaming/Cassandra use case) examining 53 metrics (6 client metrics, 5 Zookeeper metrics, 15 Kafka metrics, 5 Spark metrics, and 4 Cassandra metrics, in addition to 3 x 6 hardware resources metrics for Kafka, Spark, and Cassandra.).

[3] https://psutil.readthedocs.io/en/latest/

Each experiment is run with a stable input load for at least 10 min (not including the warm-up and the ramp-down phases, during which no measurement is performed). Each experiment is repeated 5 times and we present the average results over these runs. For all the setups that we consider, we observe a low standard deviation of the collected metrics ($< 5\%$), both during a single run and across runs with the same setup.

3.4.2 Applications

For the subsystems of the data processing pipeline including Spark Streaming, we examine three different applications described below: Wordcount, Twitter Sentiments Analysis, and Flight Delays Prediction. These applications apply different transformations on the stream of data and so they have different characteristics, one is IO intensive (Wordcount) while the others are memory intensive. For each application, the input data is ingested through Kafka, then processed by Spark Streaming, and depending on the tested case, the results are saved in Cassandra.

Note that, for the Kafka cluster use case, we use the same data as with the Wordcount application. In this case, there is no consumer subscribed to the brokers. Instead, Kafka brokers receive the messages from the clients and write them to the disk.

Wordcount (WC) is a standard micro-benchmark for big data [92]. We use Linux's dictionary to generate random corpus of English words. The WC application first reads the input data from the messaging system (Kafka), splits the words based on the space delimiter, then counts the frequency of each word and updates an in-memory map with words as the keys and current total counts as values. Finally, the results are stored either in a file or in the database depending on the examined setup.

Twitter Sentiments Analysis (TSA) monitors people's opinion on different topics. The sentiment can have a positive value (good opinion), or a negative value (bad opinion). We use the SentiWordNet [71] dictionary for opinion mining. As a dataset, we use recent tweets in English crawled through the Twitter API [25]. In this application, we process English tweets that contain words, hashtags, and mentions.

Flight Delays Prediction (FDP) uses machine learning (with a logistic regression classification algorithm) to predict the delays of airline flights. We use input data from the U.S. Department of Transportation's (DOT) Bureau of Transportation Statistics (BTS) [17].

In the next sections, we evaluate our study on three sub-systems of the data processing pipeline. We start by examining a Kafka cluster, then we add one component at a time until we build the full data processing pipeline. Thus, the three tested sub-systems are (i) Kafka cluster; (ii) Kafka/Spark Streaming cluster; (iii) Kafka/Spark Streaming/Cassandra cluster (full data processing pipeline).

3.5 Case study-1: One-tier system (Kafka cluster)

In this section, we study the Kafka cluster use case. First, we discus how to set the safety margin value M that defines the limit of the gray zone. We illustrate, in detail, the proposed approach through a concrete setup of the Kafka cluster. We apply the approach through all Kafka setups described in Table 3.10. For each setup in the Kafka cluster we strive to reply to the following questions:

- Are resource consumption metrics sufficient as indicators of performance bottleneck?

- Are there metrics that act as *reliable* indicators of performance bottleneck?
- In the case of the existence of reliable metrics, do these metrics always act as reliable indicators? Or does their behavior depend on the examined setup?

We consider different setups (more precisely 15 setups) of a Kafka cluster by changing settings that affect its performance. Table 3.10 shows the different examined setups. These setups vary between each other depending on the following factors:

- A number of partitions: In general, in a Kafka cluster the more partitions there are, the higher the throughput one can achieve[4]. We examine a different number of partitions that vary between 1, 3, 12, and 24 partitions.
- Message size: The message size setting has an impact on the performance of a Kafka cluster. Generally, in a Kafka cluster, increasing the size of the messages increases the Kafka throughput (it also leads to a higher latency). We test different values of the message size setting. These values vary between 100, 500, 1000, and 10000 bytes.
- Compression algorithm: Compressing messages sent from a Kafka client to a Kafka server (broker) affects the throughput of a Kafka cluster in its turn. We examine setups where Kafka clients send compressed messages to the Kafka brokers. We use the Snappy compression method[5].
- Acknowledgment setting: This setting has a high impact on both the system throughput and message durability. The acknowledgment setting defines how the Kafka producer knows that the message has made it to the partitions on the broker. Different values can be set for the acknowledgment setting (for more details, see Section 3.2.3).
- Hardware setup: As we have mentioned earlier, we consider different hardware configurations that vary depending on the number of the processing units (CPUs) and the type of storage (HDD and SSD).

For all setups, we monitor the evolution of the exported metrics on both software and hardware levels for both client and server sides. The goal here is to see how the exported metrics evolve when the bottleneck shifts from the client side to the server side.

3.5.1 Setting the margin value M

The threshold Th_{limit} splits the data points between two zones: (i) the zone where the client is the bottleneck; (ii) the gray zone where tuning both the client and the server can improve the overall throughput of the system. This threshold is defined relatively to the peak performance in a given setup as $Th_{limit} = Th_{0.95} \times (1 - M)$. We need to figure out what value of M could work across different setups of the Kafka cluster.

To identify the value of M for one setup, we compare the performance when using 3 and 4 Kafka brokers (deployed on 3 and 4 server nodes respectively) while increasing the number of clients. The threshold Th_{limit} for a setup corresponds to the point where the performance with 4 Kafka brokers becomes better than with 3 Kafka brokers. More precisely, we consider that using one more sever becomes beneficial if it allows increasing the throughput by at least 2%.

[4]Note that in some cases, having too many partitions may also have a negative impact. For example, more partitions requires dealing with more open file handlers in the broker. Also, more partitions may increase the end-to-end latency which is defined by the time from when a message is published by the producer to when the message is read by the consumer [32].

[5]Kafka producers support different compression methods: GZIP, LZ4, ZSTD, and Snappy. We chose Snappy as it performs faster [38]).

Setup	Num. partitions	Msg. size(B)	Compression	Acks	Platform
setup-0	12	100	no	1	Nova
setup-1	12	500	no	1	Nova
setup-2	12	1000	no	1	Nova
setup-3	12	10000	no	1	Nova
setup-4	1	100	no	1	Nova
setup-5	3	100	no	1	Nova
setup-6	24	100	no	1	Nova
setup-7	12	100	snappy	1	Nova
setup-8	3	100	no	all	Nova
setup-9	12	100	snappy	all	Nova
setup-10	12	100	no	all	Nova
setup-11	24	100	snappy	1	Nova
setup-12	12	100	no	1	Sagittaire
setup-13	12	100	no	1	Ecotype
setup-14	12	100	no	all	Ecotype

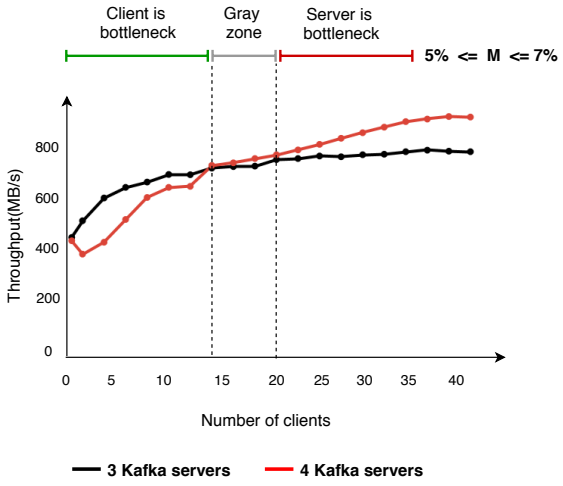
Table 3.10 – Description of the different setups of the Kafka cluster use case.

Figure 3.15 shows the system throughput as a function of the number of clients when using 3 or 4 server nodes for different setups of the Kafka cluster. The X-axis represents the number of clients and the Y-axis represents the system throughput in MB/s.

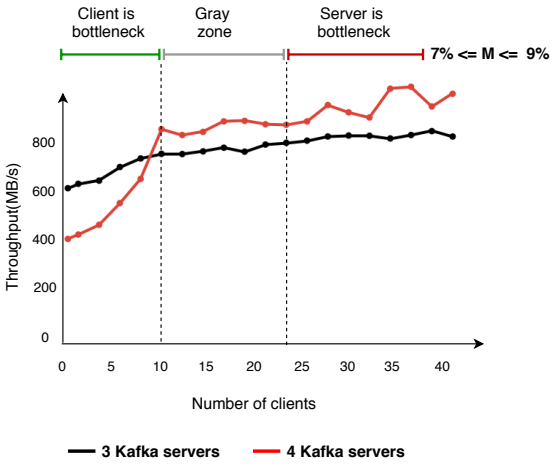
In Figure 3.15 (a), which corresponds to `setup-0`, we observe that adding one more Kafka server while increasing the number of clients does not improve the system throughput until we reach 14 clients. Thus we enter the gray zone at this point. At 14 clients, the system throughput improves by $\sim 2\%$ when using 4 Kafka brokers (738 MB/s) compared to when using 3 Kafka brokers (727 MB/s). The number of clients that corresponds to $Th_{0.95}$ (758 MB/s) is 20. It means that between 14 and 20 clients, the throughput keeps increasing while increasing the number of clients (even without adding a fourth server). Beyond 20 clients, the only way to achieve a better throughput is to add an extra server.

Based on these observations, we can deduce the possible values of M for this setup. The value of M should be chosen such that the point that corresponds to 14 clients is included in the gray zone whereas the point that corresponds to 12 clients is not. Thus, we find that the value of M for `setup-0` belongs to the range: [5%; 7%].

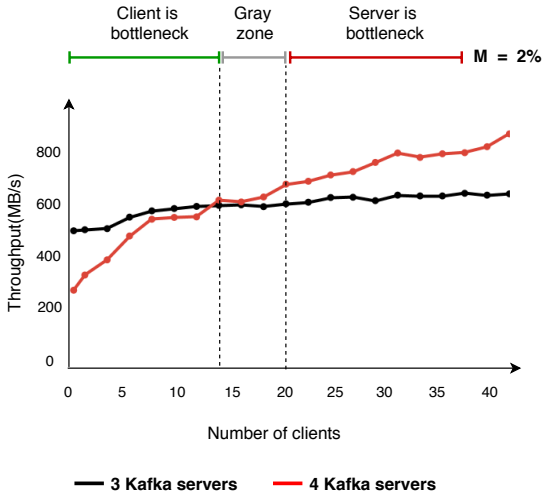
We apply the same methodology to the other setups of the Kafka cluster. Based on the different setups presented in Figure 3.15, we can observe that no single value of M allows us to precisely identify the gray zone in all setups. For the analysis of the Kafka cluster setups, we decide to select $M = 7\%$ to cover `setups-0` and `setups-2` where the throughput using 4 servers is significantly lower compared to when using 3 servers before the gray zone. `Setup-10` (Figure 3.15 (f)) is the other case where having 4 servers can have a significant negative impact before the gray zone. Note however that the error is identifying the limit of the gray zone due to setting $M = 7\%$ instead of setting $M = 4\%$ remains small in this case. The start of the gray zone is set at 16 clients instead of 20 clients.



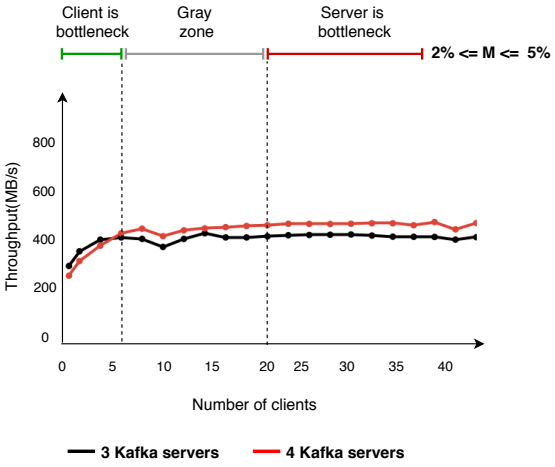
(a) Setup-0.



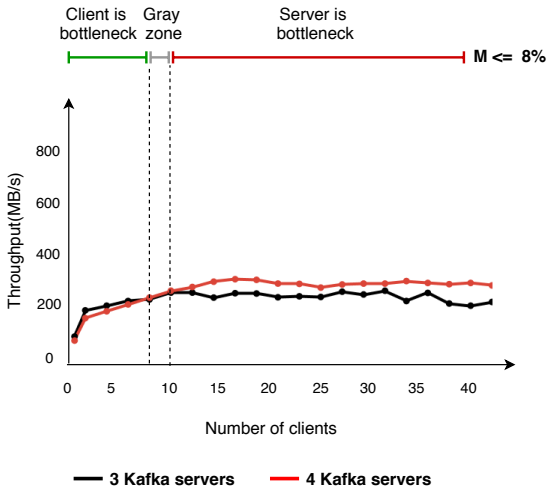
(b) Setup-2.



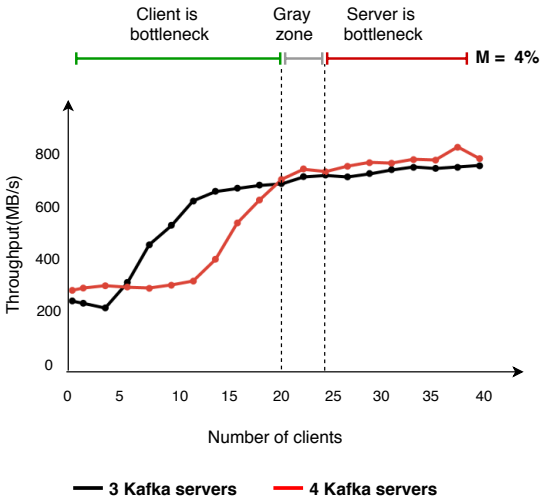
(c) Setup-6.



(d) Setup-7.



(e) Setup-9.



(f) Setup-10.

Figure 3.15 – Throughput(MB/s) of the Kafka cluster with 3 brokers vs. 4 brokers for different setups.

3.5.2 Detailed analysis of one setup

In this section, we illustrate our approach on one setup of the Kafka cluster use case. We consider `setup-0` defined in Table 3.10 and we explain in detail how we apply the proposed methodology to answer the research questions mentioned earlier. We deploy a Kafka cluster with `setup-0`, and we measure the global throughput of the system while increasing the number of clients. Figure 3.16 shows the system throughput as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the system throughput in MB/s. We compute the *peak throughput range* which is for `setup-0`: $[798 \times 0.95; 798] = [758; \quad 798]$. $Th_{0.95} = 758$ MB/s corresponds to 20 clients. As explained in the previous section, we consider $M = 7\%$ for all setups of the Kafka cluster. Thus, $Th_{limit} = Th_{0.95} \times (1 - M)$. $Th_{limit} = 758 \times 0.93 = 705$ MB/s. This corresponds to 14 clients. Based on our definition of the bottleneck, this implies that the client is the bottleneck until we reach the point where we have 14 clients injecting load. Points between 14 clients and 20 clients represent the gray zone. At 20 clients and beyond, the bottleneck shifts to the server side.

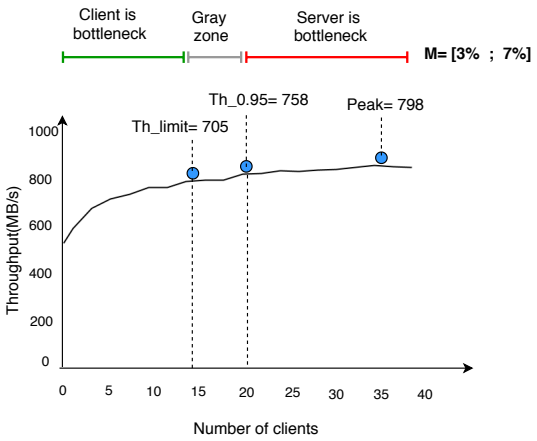


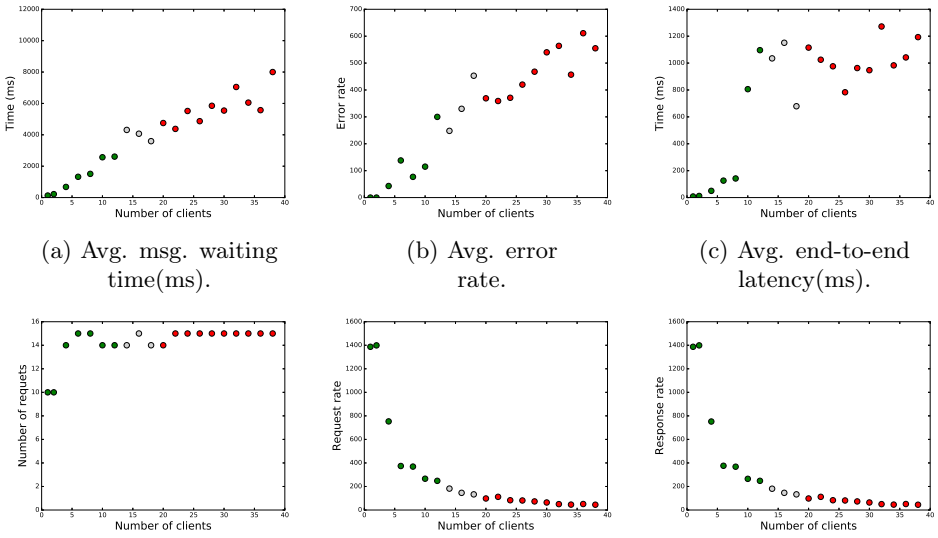
Figure 3.16 – Bottleneck identification for `setup-0` of the Kafka cluster with $M = [5\%; \quad 7\%]$.

In the following sections, we present the evolution of the metrics of both the client and the server sides for `setup-0`. We analyze the evolution of these metrics. We classify them into pattern categories as described in Section 3.3.2.6.

In all the figures presenting the evolution of a metric, the green points represent a bottleneck being on the client side. The gray points represent the gray zone where adding hardware resources to the client or the server side can improve the throughput. The red points represent a bottleneck being on the server side.

3.5.2.1 Evolution of client metrics

Figure 3.17 shows the evolution of the client metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Figures 3.17 (a), (b), and (c) follow the linear pattern. They belong to *category-3* which represents the *not reliable* metrics. Figure 3.17 (d) acts as a *misleading* metric as its behavior changes at a point that is not when the server becomes the bottleneck. Figures 3.17 (e) and (f) act as *reliable* indicators of performance bottleneck as their behaviors change at the point where the bottleneck shifts from the client side to the server side. The values of both metrics `CL_request_rate` and `CL_response_rate` decrease *linearly* while the bottleneck is on the client side and their values remain *constant* at the point where the bottleneck shifts to the server side. This change in both the metrics values and their behaviors at the point where the bottleneck shifts from one side to the other allows us to determine at any given point of the metrics values, where the bottleneck lies.



(a) Avg. msg. waiting time(ms).

(b) Avg. error rate.

(c) Avg. end-to-end latency(ms).

(d) Avg. num. in-flight requests.

(e) Avg. request rate.

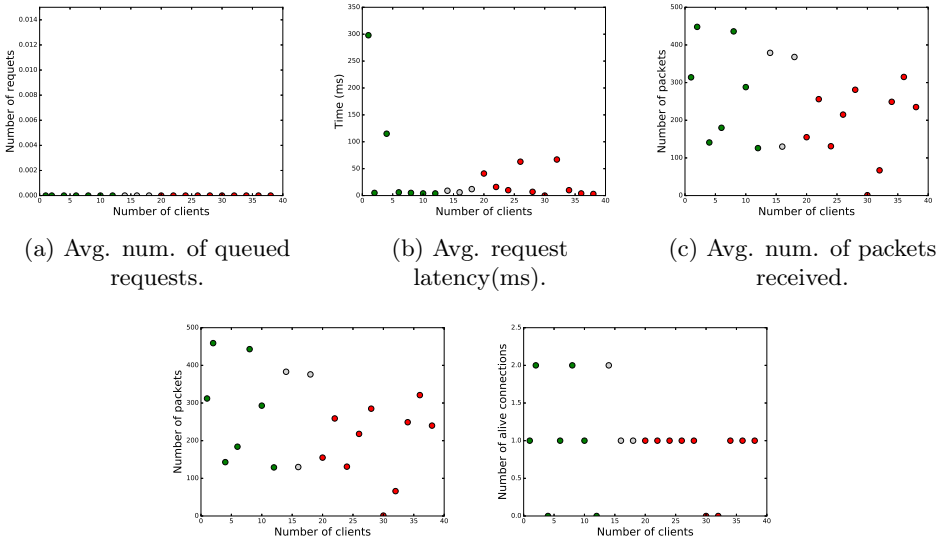
(f) Avg. response rate.

Figure 3.17 – Evolution of client software level metrics for setup-0 in the Kafka cluster use case.

3.5.2.2 Evolution of Zookeeper metrics

Figure 3.18 shows the evolution of the Zookeeper metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Analyzing the evolution of the Zookeeper metrics shows that none of the Zookeeper metrics act as a *reliable* indicator of performance bottleneck. Figures 3.18 (a), (b), (c), (d), and (e) follow patterns that fall in *category-3* which represents the *not reliable* metrics. More precisely, Figure 3.18 (a) evolves in a constant pattern where there is no change in the metric behavior nor in the values regardless of where the bottleneck lies. For Figures 3.18 (b) and (e), they evolve in a way where there is a change in their behavior but not in their values when the bottleneck component changes. Figures 3.18 (c) and (d), show random evolution of the metrics.

We examined the Zookeeper metrics for all studied cases. The results show that these metrics are always either *misleading* or *not reliable* indicators. Thus, we only show these metrics in detail for one setup of the Kafka cluster and we ignore them for the rest of the studied cases (including Kafka/Spark and Kafka/Spark/Cassandra).



(a) Avg. num. of queued requests.

(b) Avg. request latency(ms).

(c) Avg. num. of packets received.

(d) Avg. num. of packets sent.

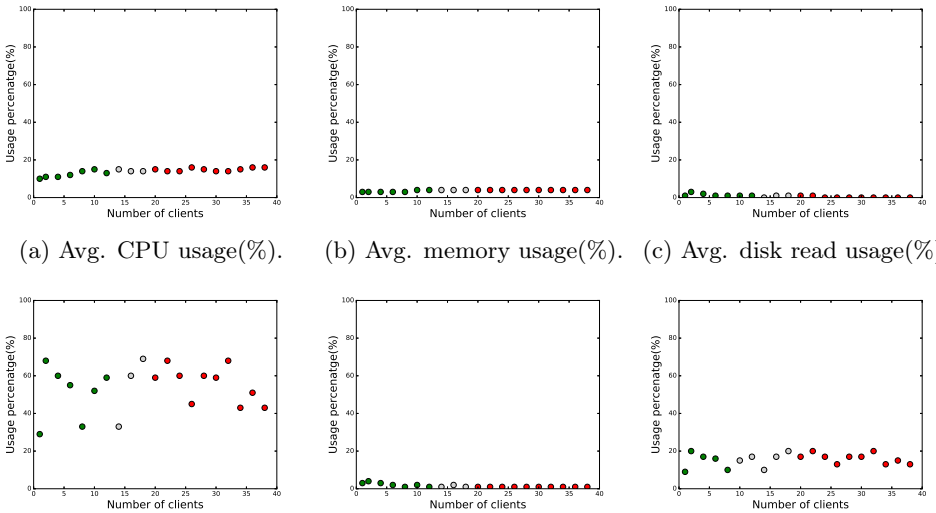
(e) Avg. num. of alive connections.

Figure 3.18 – Evolution of Zookeeper software level metrics for `setup-0` in the Kafka cluster use case.

3.5.2.3 Evolution of Kafka metrics

For Kafka metrics, we first present the evolution of resource consumption metrics, then the evolution of the software level metrics. Figure 3.19 shows the evolution of the resource consumption metrics for Kafka servers as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. We observe that the evolution of resource consumption metrics follow either constant patterns (i.e., Figures 3.19 (a), (b), (c), and (e)), or periodic variation patterns (i.e., Figures 3.19 (d) and (f)) at points corresponding to when the server reaches its maximum capacity. Consequently, resource consumption metrics are *not reliable* indicators of performance bottlenecks.

Figure 3.20 shows the evolution of the software metrics of the Kafka cluster as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Figure 3.20 (h) belongs to the *partially reliable* category as the metric evolves in a linear-periodic variation which makes it difficult to identify the limiting component in the system. The rest of the metrics belong to either *not reliable* category or the *misleading* one. Note that for the Kafka cluster case study, we do not monitor metrics that capture the consumer activities (i.e., `KF_fetchconsumer_queue_time` and `KF_fetchconsumer_total_time`). This is because there is no message consumer in the Kafka cluster use case. Instead, the Kafka servers receive the client messages and only write them to the disk.



(a) Avg. CPU usage(%). (b) Avg. memory usage(%). (c) Avg. disk read usage(%).

(d) Avg. disk write usage(%). (e) Avg. network sent usage(%). (f) Avg. network receive usage(%).

Figure 3.19 – Evolution of resource consumption metrics for `setup-0` in the Kafka cluster use case.

In the following, we strive to answer our set of research questions for the case of the Kafka cluster by applying the same analysis on the metrics for the 15 setups.

3.5.3 Are resource consumption metrics sufficient?

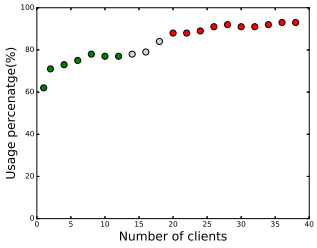
In this section, we strive to answer the question about if resource consumption metrics (i.e., CPU, memory, network, and disk IO) are sufficient indicators of a performance bottleneck. Figure 3.21 shows pie charts representing the categories of resource consumption metrics overall setups of the Kafka cluster use case. The *Unidentified* category refers to the measurements where it is not possible to identify to which pattern category the metrics belong. In `setup-14` where acknowledgment setting is set to *all* and the server is waiting for an acknowledgment from all partitions to consider a successful message, the bottleneck appears to be on the server side for the whole experiments (i.e., there are no green nor gray points in the metrics evolution graphs).

From Figure 3.21, we observe that resource consumption metrics are never *reliable* indicators of performance bottlenecks for any tested setup of the Kafka cluster use case. Hence, we cannot rely on resource consumption metrics as *reliable* indicators of performance bottleneck in the Kafka cluster use case. Several existing works [102, 117, 126] show the ability to use resource consumption metrics as indicators to detect overloaded components that could be performance bottlenecks [102, 117, 126]. These results show that such an approach does not apply to the data processing pipeline.

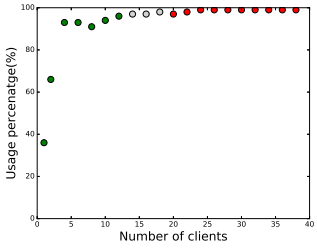
3.5.4 Are there *reliable* indicators of performance bottleneck in a Kafka cluster?

To answer our research question concerning the existence of *reliable* indicators of performance bottleneck for a given setup of a Kafka cluster, we apply the analytical study on all setups of the Kafka cluster described in Table 3.10. We summarize the output of the analytical study for all setups of the Kafka cluster in Table 3.11. Table 3.11 shows, for each tested setup, pattern categories of all metrics exported by the Kafka cluster.

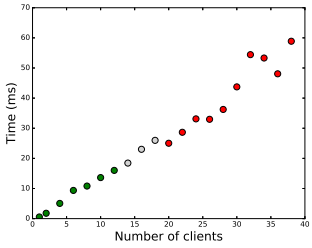
The $\{-\}$ sign in `setup-14` refers to the *Unidentified* category, i.e., we cannot infer to which category the metrics belong as explained earlier.



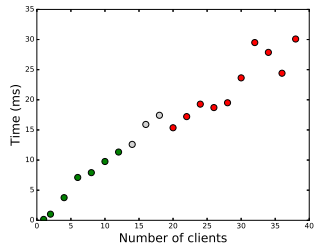
(a) Avg. network threads usage(%).



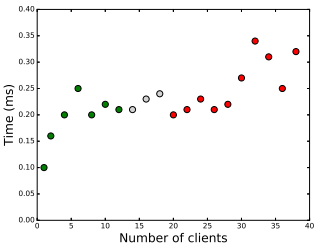
(b) Avg. handler threads usage(%).



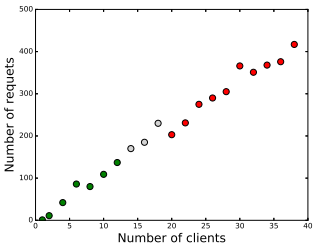
(c) Avg. response queue time(ms).



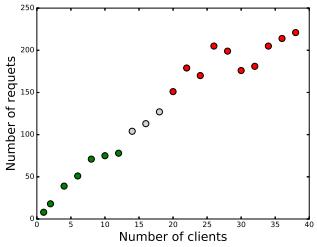
(d) Avg. request queue time(ms).



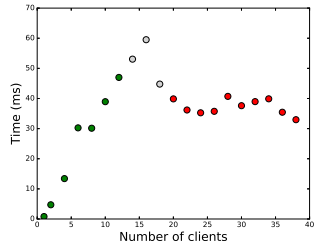
(e) Avg. request processing time(ms).



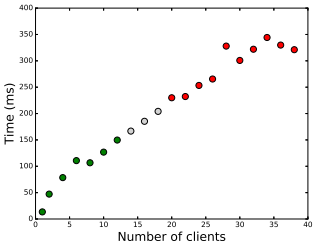
(f) Avg. request queue size.



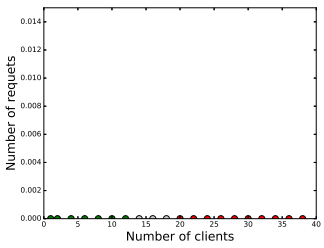
(g) Avg. response queue size.



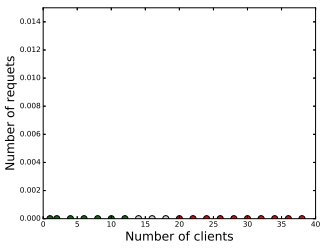
(h) Avg. fetchfollower queue waiting time(ms).



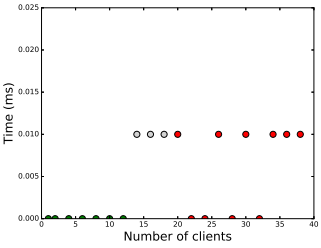
(i) Avg. fetchfollower total time(ms).



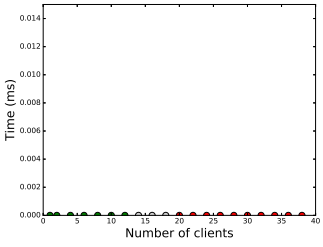
(j) Avg. producer purgatory size.



(k) Avg. fetchfollower purgatory size.



(l) Avg. fetchfollower throttle time(ms).



(m) Avg. producer throttle time(ms).

Figure 3.20 – Evolution of Kafka software level metrics for setup-0 in the Kafka cluster use case.

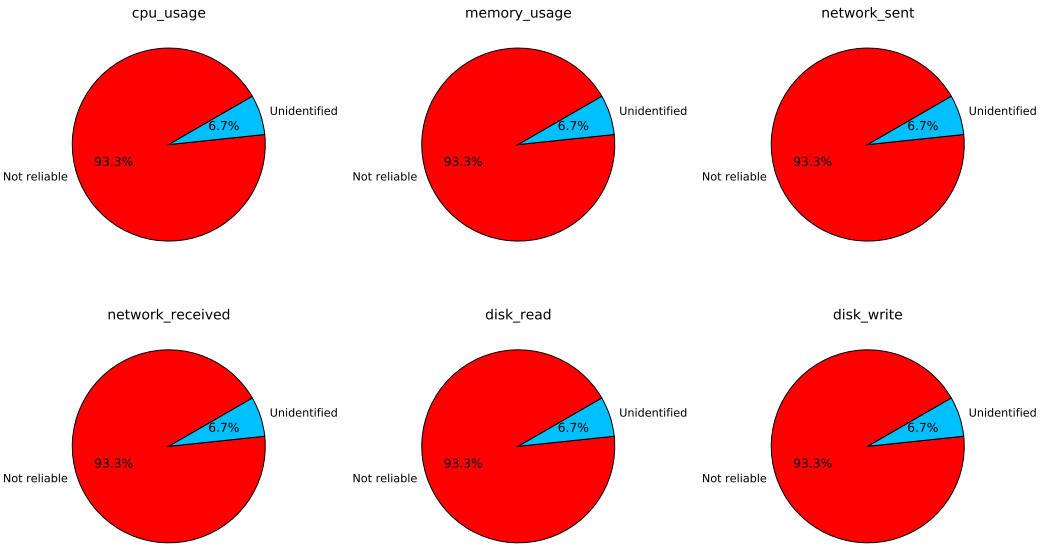


Figure 3.21 – The pattern category (in percent) that Kafka resource consumption metrics follow for all setups of the Kafka cluster use case.

From Table 3.11, we observe that:

- For a given setup of the Kafka cluster, there is always at least one *reliable* metric that we can rely on in identifying the performance bottleneck in the system.
- There are always client metrics that act as *reliable* indicators of performance bottleneck for any given setup of the Kafka cluster.
- All client metrics appear to be *reliable* indicators in at least one setup of the Kafka cluster. However, it is not the case for the server (Kafka brokers) metrics where some metrics are **never** *reliable* indicators for any tested setup (e.g., `producer_throttletime`, and `KF_fetchfollower_purgatorySize`).
- To cover all the setups of the Kafka cluster, at least one metric from the client side combined with at least one metric from the server side are sufficient to identify performance bottlenecks. The client metric should be one of these two metrics: `CL_request_rate` or `CL_response_rate`, while for the server metric, it should be one of these 4 metrics:
  - `KF_network_threads_usage`;
  - `KF_handler_threads_usage`;
  - `KF_request_processing_time`;
  - `KF_producer_purgatorySize`.

  Figure 3.23 shows a possible combination of reliable client and server metrics to cover all setups of the Kafka cluster. Also, it shows the types of these metrics as described in Table 3.2. Based on Figure 3.23, we can combine two groups of *reliable* metrics. We use one metric per group to cover all tested setups of the Kafka cluster. More precisely, the combination should include a metric that belongs to the **In/Out Data** type and a metric that belongs to the **Processing Time** type.

- We can rely on server (Kafka brokers) metrics to identify the bottleneck component for most setups of the Kafka cluster. Relying on, at least, two metrics on the server side can cover most setups of the Kafka cluster (e.g., combining the two Kafka metrics: `KF_network_threads_usage` and `KF_fetchfollower_queue_time` can cover most setups of the Kafka cluster). Note that for `setup-0`, there are *only* client metrics that act as reliable indicators of performance bottleneck while none of the server metrics do. To confirm this point, we show the evolution of the metrics for another setup of the Kafka cluster where some server metrics act as *reliable* indicators of performance bottleneck. Figure 3.22 shows the Kafka metrics evolution for `setup-7`. From Figure 3.22, we observe that some server metrics act as reliable indicators of performance bottleneck. `KF_response_queue_time` (Figure 3.22 (c)), `KF_request_queue_time` (Figure 3.22 (d)), `KF_request_processing_time` (Figure 3.22 (e)), `KF_fetchfollower_queue_time` (Figure 3.22 (h)), and `KF_fetchfollower_total_time` (Figure 3.22 (i)) act as reliable indicators of performance bottleneck. So, to identify the bottleneck component in `setup-7`, metrics exported by the server side are sufficient.

  Figure 3.24 shows a possible combination of server-side-only metrics that are sufficient to cover 13 out of 14 setups (we exclude the `setup-14` as we cannot infer the pattern category that its metrics follow) of the Kafka cluster. In this combination, we need one metric that belongs to the **Idle Threads** type with another metric that belongs to the **Queue Waiting Time** type. Other possible combinations can be found in Table 3.11.

3.5.5 What are the characteristics of performance bottleneck indicators?

After we have seen that there are metrics that act as *reliable* indicators of performance bottleneck in a given setup of the Kafka cluster, in this section, we strive to answer the research question concerning the characteristics of these indicators (i.e., the category that these indicators belong to). Depending on the tested setup, the category to which a metric belongs may change. Figure 3.26 shows pie charts representing, for each client metric, the percentage to each category a metric belongs depending on the setup. Figure 3.27 shows pie charts summarizing, for each Kafka metric, the percentage to each category a metric belongs depending on the setup. Figures 3.26 and 3.27 represent the data of Table 3.11 in a different way.

From both Figures 3.26 and 3.27, we observe that:

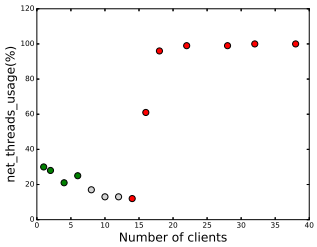
- On the client side, metrics like `CL_error_rate`, `CL_request_in_flight`, `CL_request_rate`, and `CL_response_rate` act as *reliable* indicators of performance bottleneck for most setups. These metrics belong to the following types (described in Table 3.2): **Error Rate**, **Queue Size**, and **In/Out Data**.

- On the server side, metrics like `KF_network_threads_usage`, `KF_handler_threads_usage`, `KF_request_queue_time`, and `KF_fetchfollower_queue_time` act as *reliable* indicators of performance bottleneck for most setups. These metrics belong to the following types: **Idle Threads** and **Queue Waiting Time**.

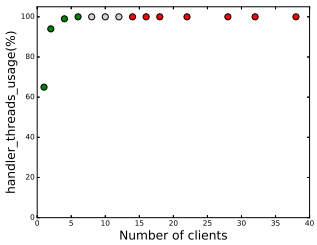
- For both client and server sides, some metrics that act as *reliable* indicators of performance bottlenecks for most setups, sometimes act as *misleading* indicators (e.g., `CL_error_rate` on the client side and `KF_handler_threads_usage` on the server side). More specifically, on the client side, `CL_request_in_flight`, `CL_request_rate`, and `CL_response_rate` act as *misleading* indicators in 6.7% of the cases which represents 1/15 setups and the `CL_error_rate` acts as a *misleading* in

Metrics	Setups														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Client metrics															
CL_msg_queue	NR	NR	R	NR	NR	NR	PR	R	NR	R	NR	R	NR	NR	-
CL_error_rate	NR	R	R	R	NR	R	NR	M	R	M	R	R	NR	R	-
CL_request_latency	NR	R	R	NR	NR	NR	NR	R	NR	R	NR	R	NR	NR	-
CL_request_in_flight	NR	R	R	R	NR	PR	NR	PR	M	NR	PR	R	PR	R	-
CL_request_rate	R	R	R	R	R	R	R	R	R	M	NR	R	R	R	-
CL_response_rate	R	R	R	R	R	R	R	R	R	M	NR	R	R	R	-
Kafka metrics															
cpu_usage	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	-
memory_usage	NR	M	NR	NR	NR	NR	NR	M	NR	NR	NR	NR	NR	NR	-
network_sent	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	-
network_received	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	-
disk_read	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	-
disk_write	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	-
KF_network_threads_usage	NR	NR	R	R	R	R	R	PR	R	M	R	R	NR	R	-
KF_handler_threads_usage	M	R	R	R	R	M	NR	M	R	M	R	R	NR	R	-
KF_response_queue_time	NR	NR	R	NR	NR	NR	NR	R	NR	R	NR	R	R	NR	-
KF_request_queue_time	NR	R	R	R	NR	NR	NR	R	R	R	NR	NR	R	R	-
KF_request_processing_time	NR	PR	NR	PR	NR	R	NR	R	NR	R	R	NR	PR	NR	-
KF_request_queue_size	NR	NR	NR	PR	NR	NR	NR	M	R	M	NR	R	NR	R	-
KF_response_queue_size	NR	NR	NR	NR	NR	NR	NR	M	M	M	NR	R	M	NR	-
KF_fetchfollower_queue_time	PR	R	R	R	NR	R	PR	R	PR	R	PR	R	R	R	-
KF_fetchfollower_total_time	M	NR	NR	NR	NR	NR	NR	R	NR	NR	NR	R	NR	R	-
KF_producer_purgatorySize	NR	NR	NR	NR	NR	NR	NR	NR	NR	M	R	NR	NR	NR	-
KF_fetchfollower_purgatorySize	NR	NR	NR	NR	NR	NR	NR	NR	NR	PR	NR	NR	NR	NR	-
KF_fetchfollower_throttletime	NR	R	M	PR	NR	NR	M	NR	NR	NR	NR	NR	NR	NR	-
KF_producer_throttletime	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	M	NR	-

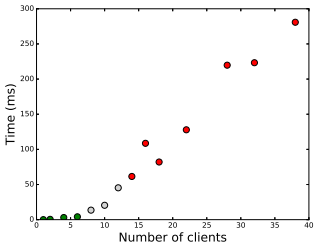
Table 3.11 – Analytical study results for all setups of the Kafka cluster (R: reliable, PR: partially reliable, M: misleading, and NR: not reliable).



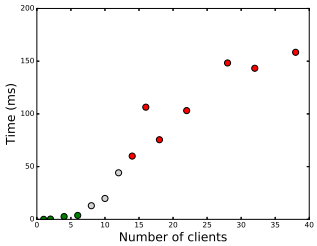
(a) Avg. network threads usage(%).



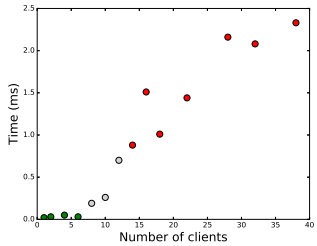
(b) Avg. handler threads usage(%).



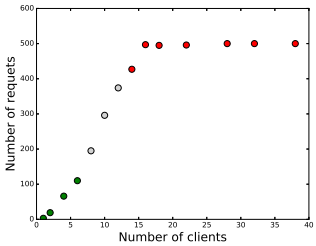
(c) Avg. response queue time(ms).



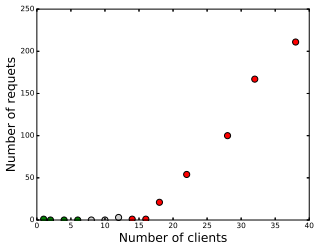
(d) Avg. request queue time(ms).



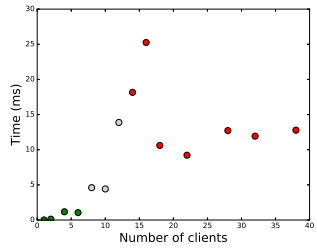
(e) Avg. request process time(ms).



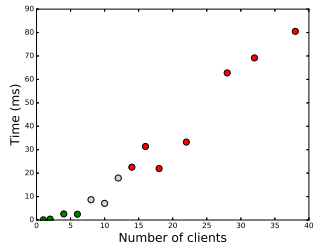
(f) Avg. request queue size.



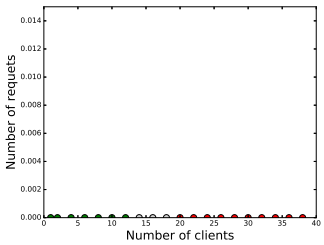
(g) Avg. response queue size.



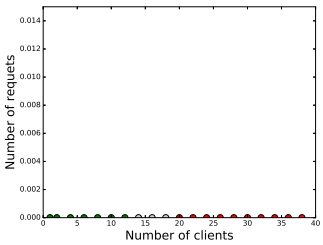
(h) Avg. fetchfollower queue waiting time(ms).



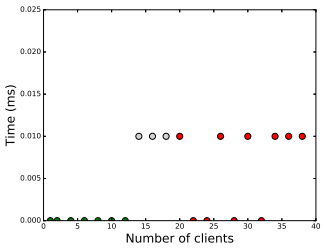
(i) Avg. fetchfollower total time(ms).



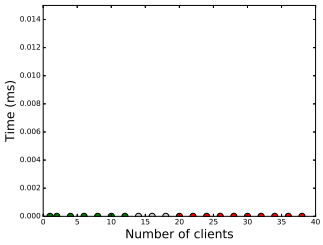
(j) Avg. producer purgatory size.



(k) Avg. fetchfollower purgatory size.



(l) Avg. fetchfollower throttle time(ms).



(m) Avg. producer throttle time(ms).

Figure 3.22 – Evolution of Kafka metrics for setup-7.

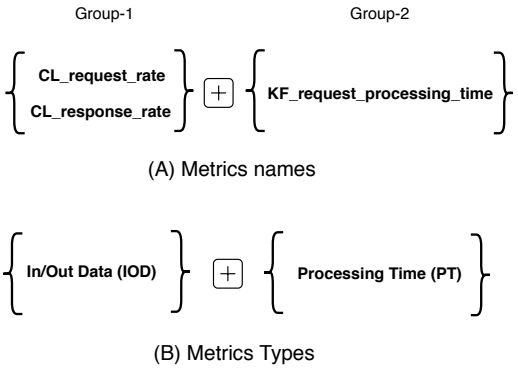


Figure 3.23 – A possible combination of *reliable* indicators of performance bottleneck for the Kafka cluster use case.

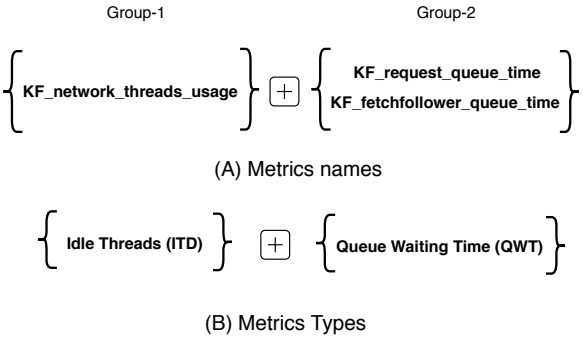


Figure 3.24 – A possible combination of *reliable* server-side-only metrics for the Kafka cluster use case.

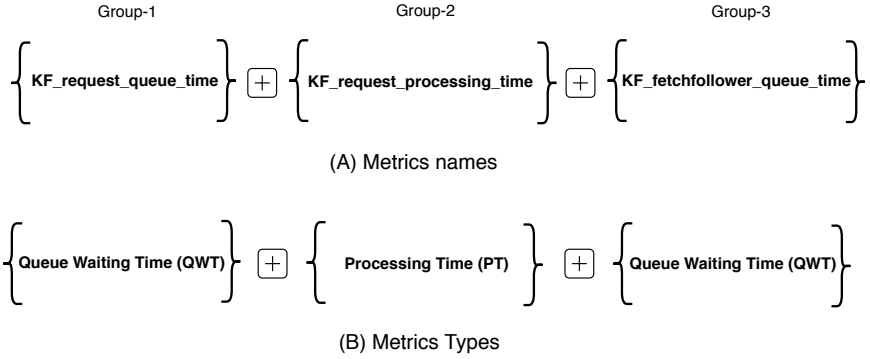


Figure 3.25 – A combination of *reliable* indicators that are *never* misleading for the Kafka cluster use case.

13.3% of the cases which represents 2/15 setups. On the server side, `KF_network_threads_usage` and `KF_handler_threads_usage` act as *reliable* indicators in 60% and 53% of the setups respectively. However, the `KF_network_threads_usage` metric acts as *misleading* in 6.7% of the cases which represents 1/15 setups. The `KF_handler_threads_usage` metric acts as *misleading* in 26.7% of the cases which represents 4/15 setups. Relying on these metrics might lead to wrong decisions, i.e., they might lead to a false identification of a bottleneck on the server side while it is not and consequently, we might over-provision hardware resources to the server while it is not needed.

Based on the previous observations, it should be noted that all combinations of metrics presented in Figures 3.23 and 3.24 include at least one metric that becomes misleading on some setups. It means building a tool that always automatically identifies the bottleneck correctly might be difficult using these metrics. Figure 3.25 shows that selecting 3 server metrics (`KF_request_queue_time`, `KF_request_processing_time`, and `KF_fetchfollower_queue_time`) is the best combination of metrics including metrics that are never misleading. With these 3 metrics, we can identify the bottleneck in 11 over 14 cases.

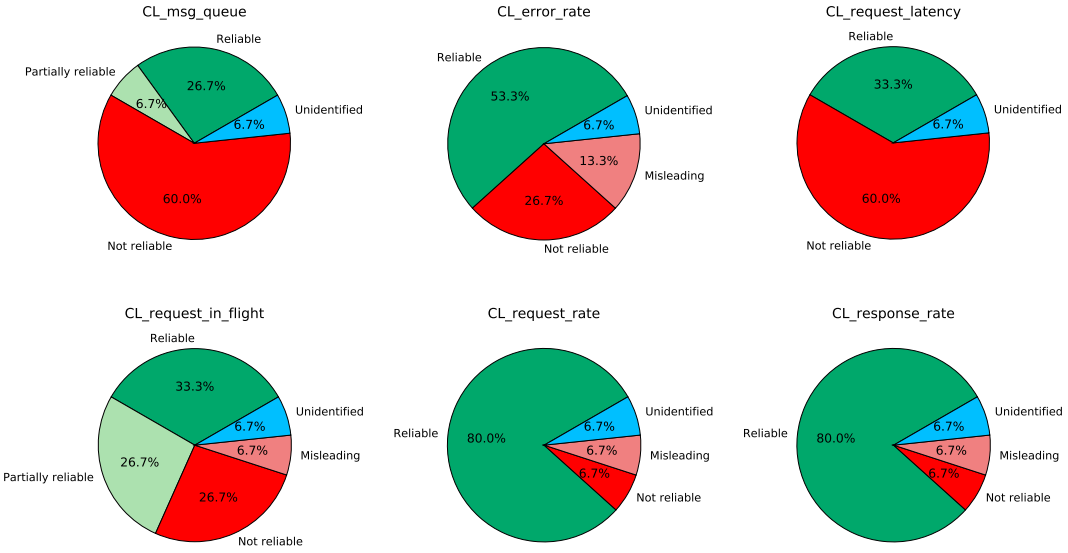


Figure 3.26 – The pattern category (in percent) that client metrics follow for all setups of the Kafka cluster use case.

3.5.6 Discussion

Based on the experimental study on the Kafka cluster use case with different setups, we summarize our findings as follows:

- Unlike many existing works [102, 117, 126] where resource consumption metrics are used as indicators to detect overloaded components that could be performance bottlenecks, resource consumption metrics are never *reliable* indicators of performance bottleneck in any tested setup of the Kafka cluster.
- For each setup of the Kafka cluster, there are always metrics that act as *reliable* indicators of performance bottleneck. This answers our first question about the existence of *reliable* indicators of performance bottleneck for a given setup of the studied system.

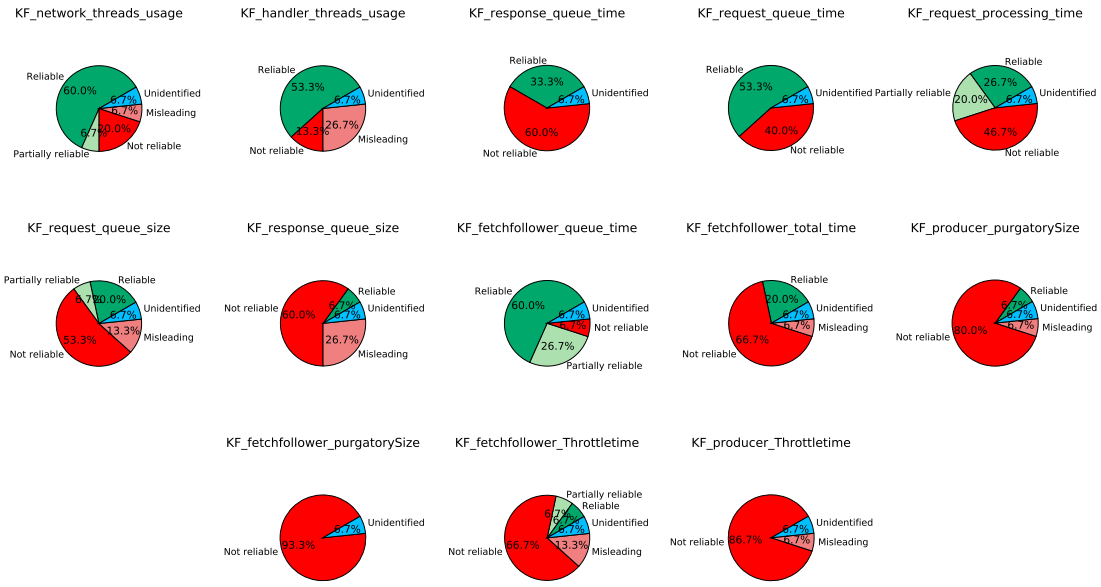


Figure 3.27 – The pattern category (in percent) that each Kafka metric follows for all setups of the Kafka cluster use case.

- Metrics belonging to the **Queue Waiting Time** type (e.g., `KF_request_queue_time`) participate in being *reliable* indicators of performance bottleneck in the Kafka cluster. These metrics always appear to be combined with other client or server metrics to cover most setups.
- Metrics belonging to the **Latency** type (e.g., `CL_request_latency`), act as *not reliable* indicators for most setups. This type of metric does not strongly participate in the combination of metrics to identify the limiting component in the system. Unlike the existing works [73, 108, 150] where the authors rely *only* on latency metrics to identify a bottleneck component in their systems, our study does not show the **Latency** category contributing to identifying performance bottleneck in the Kafka cluster use case.
- A combination of reliable metrics on both client and server sides is sufficient to cover all cases of the Kafka cluster. As we have seen, combining one of these two client metrics (i.e., `CL_request_rate` and `CL_response_rate`) with one or two of these server metrics (i.e., `KF_network_threads_usage`, `KF_handler_threads_usage`, `KF_request_processing_time`, and `KF_producer_purgatorySize`) is sufficient to identify the performance bottleneck for any of the tested setups of the Kafka cluster.
- It is possible to rely only on metrics exported on the server side to identify the bottleneck component in a Kafka cluster. For example, combining the two Kafka metrics, `KF_network_threads_usage`, and `KF_fetchfollower_queue_time`, is sufficient to identify the bottleneck component for most setups of the Kafka cluster.
- It is possible to rely on metrics that are never misleading for any setup of the Kafka cluster. For example, a combination of 3 server metrics (i.e., `KF_request_queue_time`, `KF_request_processing_time`, and `KF_fetchfollower_queue_time`) is sufficient to identify the bottleneck component for most setups of the Kafka cluster.

- In general, for the Kafka use case, metrics that belong to these types: **Error Rate**, **Queue Size**, **In/Out Data**, **Idle Threads** and **Queue Waiting Time** are *reliable* indicators of performance bottleneck.

Table 3.12 summarizes the output of the analytical study for all tested setups of the Kafka cluster.

Metric Type	Kafka cluster	
	Client	Server
Resource Consumption (RC)		
Idle Threads (ITD)		✓
Error Rate (ER)		
Queue Waiting Time (QWT)		✓
Queue Size (QS)		✓
Latency (LY)		
Processing Time (PT)		✓
In/Out Data (IOD)	✓	
Uncategorized (UC)		

Table 3.12 – Summary of reliable metrics types for all setups of the Kafka cluster use case.

In the next section, we apply the proposed methodology onto a subsystem where the server is composed of two components: Kafka and Spark Streaming.

3.6 Case study-2: Two-tier system (Kafka/Spark cluster)

In this section, we study a stack comprising two main components: Kafka and Spark Streaming. We illustrate the approach in detail on one setup of the Kafka/Spark cluster case study. Throughout the study, we try to answer the research questions concerning the existence of *reliable* indicators of performance bottleneck and their characteristics. We discuss how fine-grained the conclusions that we can draw from these indicators are (i.e., in the case of the server being the bottleneck, can the indicators tell whether Kafka or Spark is the limiting component?). Finally, we summarize and highlight the main findings of the study for all setups of the Kafka/Spark cluster.

We consider 10 setups of the Kafka/Spark use case (see Table 3.13). For all setups, we deploy the Kafka instances on 3 dedicated nodes and the Spark instances on 3 other nodes.

We examine three different applications: Wordcount (WC), Twitter sentiment analysis (TSA), and Flight delay prediction (FDP). For all setups, we consider a 100B message size, no compression, and acknowledgment option equals to 1. All setups are performed on the Nova cluster (see Table 3.9). These setups vary between each other depending on the following factors:

- The number of Kafka partitions: In a Kafka/Spark cluster, the number of Kafka partitions defines the parallelism level of reading the input data by Spark executors from Kafka partitions. We use the Direct Stream approach [34]. This approach provides simple parallelism correspondence between Kafka partitions and Spark partitions. We consider different number of Kafka partitions that varies between 12, 24, and 36 partitions.

- The number of Spark executors: As mentioned earlier Spark executors are worker nodes processes in charge of running individual tasks in a given Spark job. Generally, increasing the number of Spark executors affect system performance. We evaluate different numbers of Spark executors that vary

between 3, 12, and 18. Consequently, it is useless to set a number of Spark executors higher than the number of Kafka partitions.

- Workload type: We examine different kinds of workload applications (see Section 3.4.2). These applications vary in their characteristics between applying simple operations on the input data (i.e., Wordcount) to using machine learning techniques (i.e., Flight delay prediction).

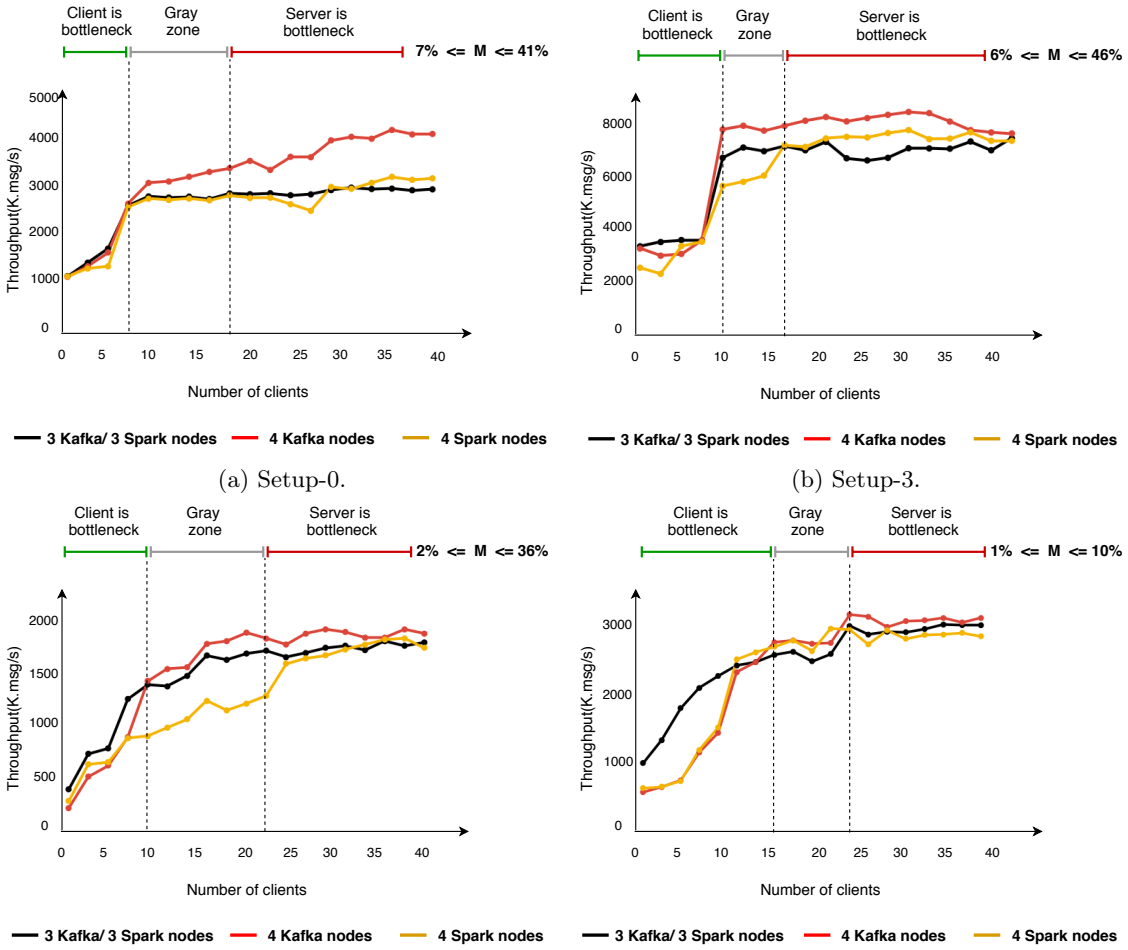
Setup	Num. Kafka partitions	Num. Spark executors	Workload
`setup-0`	12	3	WC
`setup-1`	12	12	WC
`setup-2`	24	18	WC
`setup-3`	36	12	WC
`setup-4`	12	3	TSA
`setup-5`	12	12	TSA
`setup-6`	24	18	TSA
`setup-7`	12	3	FDP
`setup-8`	12	12	FDP
`setup-9`	24	18	FDP

Table 3.13 – Description of the different examined setups for the Kafka/Spark use case.

3.6.1 Setting the margin value M

In this section, we apply the methodology described in Section 3.5.1 for setting the value of M in the Kafka/Spark case study. The goal of finding M is to define the gray zone boundaries. More precisely, we want to find the lower-bound of the gray zone (the upper-bound is defined based on $Th_{0.95}$). Figure 3.28 shows the system throughput as a function of the number of clients for different setups of the Kafka/Spark use case. Each sub-figure shows the system throughput with 3 and 4 nodes for each component in the system (i.e., Kafka and Spark). The X-axis represents the number of clients and the Y-axis represents the system throughput in K.msg/s. The black curve represents the system throughput when 3 nodes are used for each component. The red curve represents the system throughput when 4 nodes are used for Kafka instances in the system (while 3 nodes are used for Spark instances). The yellow curve represents the system throughput when 4 nodes are used for Spark instances (while 3 nodes are used for Kafka instances).

Based on Figure 3.28, we observe that increasing the number of Kafka instances (nodes) increases the system throughput at a certain number of clients. However, increasing the number of Spark nodes mostly does not improve the system throughput. Hence, the beginning of the gray zone is when having 4 Kafka instances starts providing better performance. We present the corresponding ranges of values of M that can allow identifying the beginning (lower-bound) of the gray zone for the different setups. Taking the intersections of these ranges, we observe that $M = 7\%$ is again a good choice to identify the beginning of the gray zone.



(a) Setup-0.

(b) Setup-3.

(c) Setup-7.

(d) Setup-9.

Figure 3.28 – Throughput(K.msg/s) of a Kafka/Spark cluster with 3 Kafka/Sparks nodes vs. 4 Kafka/Sparks nodes for different setups.

3.6.2 Detailed analysis of one setup

In this section, we illustrate our methodology through a concrete setup of the Kafka/Spark use case. We consider `setup-0` defined in Table 3.13. We measure the global throughput of the system while increasing the number of clients. Figure 3.29 shows the system throughput as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the system throughput. We compute the *peak throughput range* which is in our case: $[2875 \times 0.95; \quad 2875] = [2731; \quad 2875]$. $Th_{0.95} = 2731$ K.msg/s corresponds to 18 clients. $Th_{limit} = 2458$ K.msg/s corresponds to 10 clients. Based on our definition of the bottleneck, the client is the bottleneck until we reach 10 clients injecting load. Points between 10 clients and 18 clients represent the gray zone. At 18 clients and beyond, the bottleneck shifts to the server side (either to Kafka or to Spark).

In the following sections, we present the evolution of the metrics for both client and server sides. We analyze the evolution of these metrics. We want to classify them into pattern categories as described in Section 3.3.2.6. The results show that resource consumption metrics always evolve constantly for this setup, thus, we do not show the evolution of these metrics. In all the figures presenting the evolution of a metric, the green points represent a bottleneck being on the client side. The gray points represent the gray zone where adding hardware resources to the client or the server side can improve the throughput. The red points

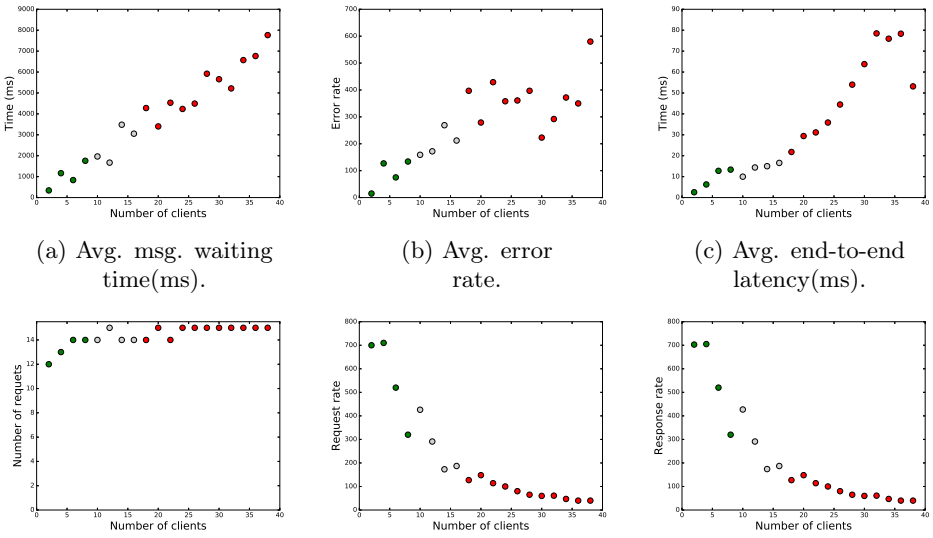
Figure 3.29 – Bottleneck identification for setup-0 of the Kafka/Spark cluster.

represent a bottleneck being on the server side (Kafka or Spark).

3.6.2.1 Evolution of client metrics

Figure 3.30 shows the evolution of the client metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Base on Figure 3.30, we observe that:

- The `CL_error_rate` metric seems to belong to the *reliable* category as it seems to change its values and its behavior at the point where the bottleneck shifts from the client side to the server side.
- The `CL_request_in_flight` metric acts as a *misleading* indicator as the metric does not change its behavior at the point where the bottleneck shifts from the client side to the server side.
- The following metrics: `CL_msg_queue`, `CL_request_latency`, `CL_request_rate`, and `CL_response_rate` act as *not reliable* indicators as their values evolve in a close to the linear way.



(a) Avg. msg. waiting time(ms).

(b) Avg. error rate.

(c) Avg. end-to-end latency(ms).

(d) Avg. num. in-flight requests.

(e) Avg. request rate.

(f) Avg. response rate.

Figure 3.30 – Evolution of client software level metrics for setup-0 of the Kafka/Spark use case.

3.6.2.2 Evolution of Kafka metrics

Figure 3.31 shows the evolution of Kafka metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Figures 3.31 (a) and (i) show evolutions of metrics that belong to the *reliable* category. Figures 3.31 (b) and (h) show evolution of metrics that belong to the *partially reliable* category. The rest of the metrics act as *not reliable* indicators.

3.6.2.3 Evolution of Spark metrics

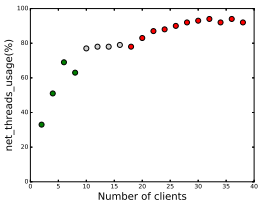
Figure 3.32 shows the evolution of Spark metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Figures 3.32 (a), (c), (d), and (e) show evolution of *not reliable* indicators. Figure 3.32 (b) shows the evolution of `SP_processing_time` metric which acts as *reliable* indicator. Although it could also be considered as a not reliable indicator, we prefer to classify `SP_processing_time` metric as a reliable indicator since we can observe a change in its behavior at 10 clients and since the value of the metric with more than 10 clients is always higher than with 10 clients.

In the following sections, we first check if the resource consumption metrics (for both Kafka and Spark) are sufficient for identifying the performance bottleneck for a given setup of a two-tier processing pipeline. Then, we look into the exported metrics by all the system components to verify if there are *reliable* indicators of performance bottleneck. In the case of the existence of these *reliable* indicators, we try to identify their characteristics. Finally, we summarize and discuss the study findings.

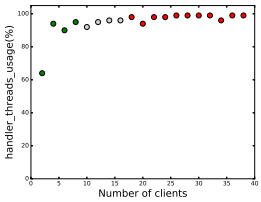
3.6.3 Are resource consumption metrics sufficient?

Similarly to the Kafka cluster use case, in this section, we strive to answer the question about resource consumption metrics (i.e., CPU, memory, and IO) being sufficient indicators of performance bottleneck. We check resource consumption metrics for both server components: Kafka and Spark.

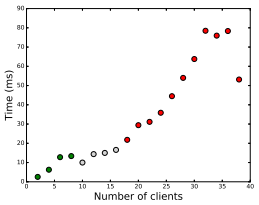
Figures 3.33 and 3.34 show pie charts representing the categories of resource consumption metrics for all setups for Kafka and Spark respectively. The *Unidentified* category refers to the measurements where it is



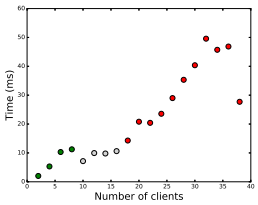
(a) Avg. network threads usage(%).



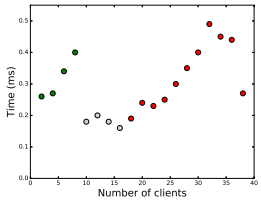
(b) Avg. handler threads usage(%).



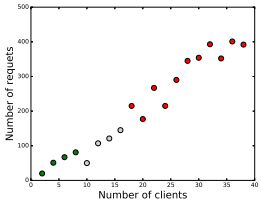
(c) Avg. response queue time(ms).



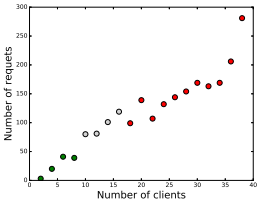
(d) Avg. request queue time(ms).



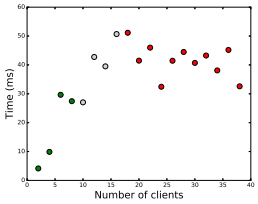
(e) Avg. request process time(ms).



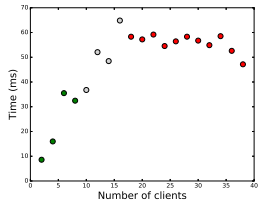
(f) Avg. request queue size.



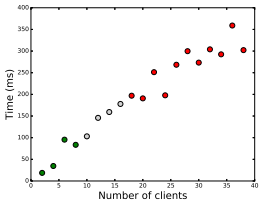
(g) Avg. response queue size.



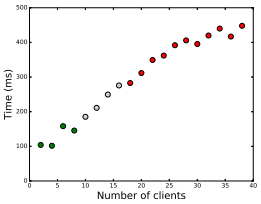
(h) Avg. fetchfollower queue waiting time(ms).



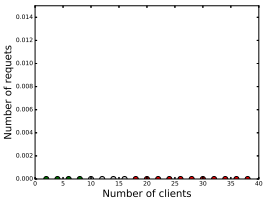
(i) Avg. fetchconsumer queue waiting time(ms).



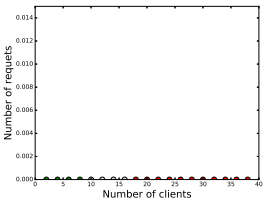
(j) Avg. fetchfollower total time(ms).



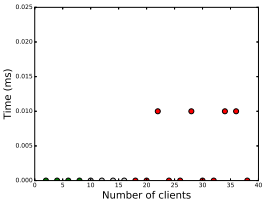
(k) Avg. fetchconsumer total time(ms).



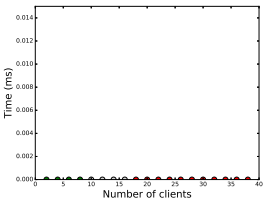
(l) Avg. producer purgatory size.



(m) Avg. fetchfollower purgatory size.

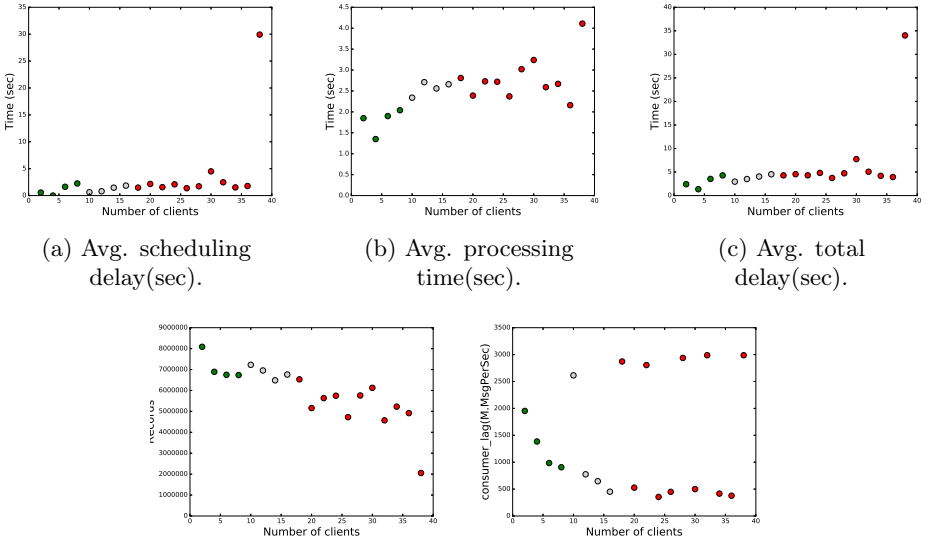


(n) Avg. fetchfollower throttle time(ms).



(o) Avg. producer throttle time(ms).

Figure 3.31 – Evolution of Kafka software level metrics for setup-0 of the Kafka/Spark use case.



(a) Avg. scheduling delay(sec).

(b) Avg. processing time(sec).

(c) Avg. total delay(sec).

(d) Avg. number of records.

(e) Avg. consumer lag(M.msg).

Figure 3.32 – Evolution of Spark software level metrics for setup-0 of the Kafka/Spark use case.

not possible to identify to which pattern category the metrics belong. This happens for instance because the bottleneck appears to be on the server side for the duration of the whole experiments (i.e., there are no green nor gray points in the metrics evolution graphs).

Based on Figures 3.33 and 3.34, we observe that resource consumption metrics are never *reliable* indicators of performance bottlenecks for any tested setup of the Kafka/Spark cluster use case. Hence, we cannot rely on resource consumption metrics as *reliable* indicators of performance bottleneck in the Kafka/Spark cluster use case.

3.6.4 Are there *reliable* indicators of performance bottleneck in a Kafka/Spark cluster?

To answer the question concerning the existence of *reliable* indicators of performance bottleneck for a given setup of a Kafka/Spark cluster, we apply the analytical study on all setups of Kafka/Spark cluster described in Table 3.13. Table 3.14 shows, for each tested setup of the Kafka/Spark cluster, pattern categories of all metrics exported by a Kafka/Spark cluster.

The $\{-\}$ sign in `setup-4` refers to the *Unidentified* category, i.e., we cannot infer to which category the metrics belong. This is because the bottleneck appears to be on the server side for the duration of the whole experiments. Even worse in this setup, the *global throughput* of the system decreases while increasing the injected load.

From Table 3.14, we observe that:

- For any given setup of a Kafka/Spark cluster, there is always, at least, either one *reliable* or one *partially reliable* metric of performance bottleneck. For `setup-7` there are *partially reliable* indicators but no reliable ones.
- There are always server metrics that act as *reliable* indicators of performance bottleneck for any setup (except for `setup-7`). More precisely they are Kafka metrics.

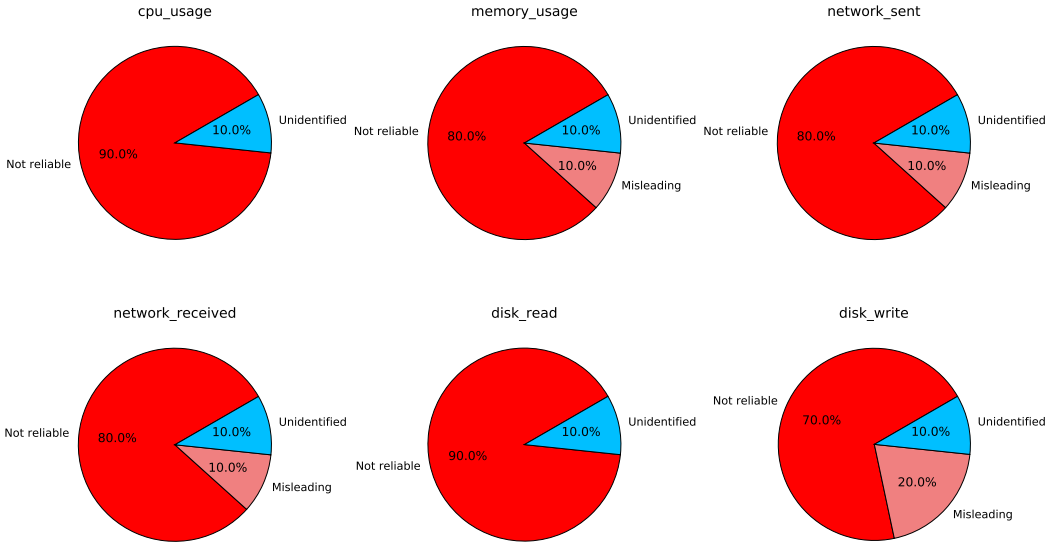


Figure 3.33 – The pattern category (in percent) that Kafka resource consumption metrics follow for all setups of the Kafka/Spark cluster case study.

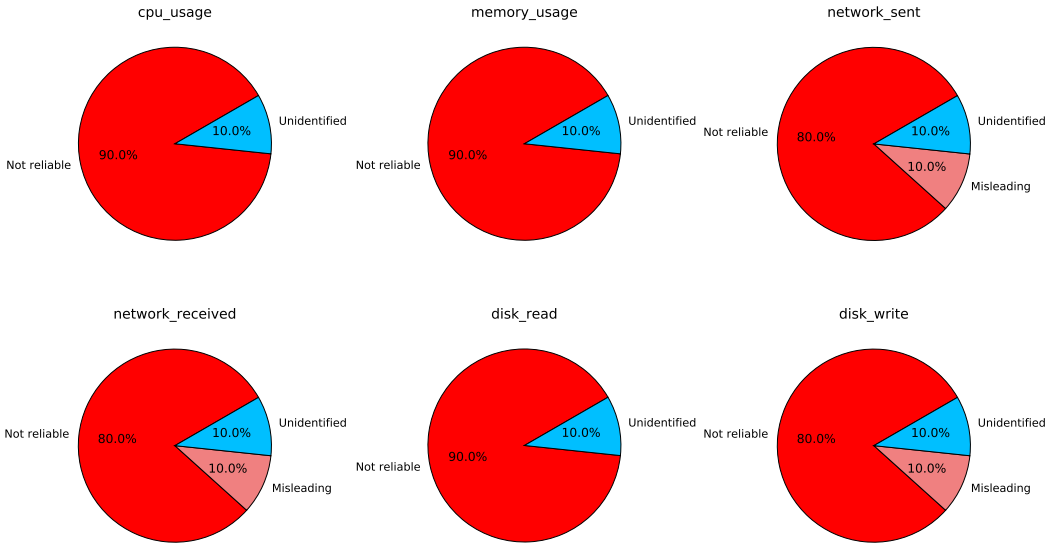


Figure 3.34 – The pattern category (in percent) that Spark resource consumption metrics follow for all setups of the Kafka/Spark cluster case study.

- There is a set of metrics that we can rely on to cover all setups of the Kafka/Spark cluster. For example, combining a client metric (i.e., `CL_error_rate`) with a Kafka metric (i.e., `KF_fetchconsumer_queue_time`) and a Spark metric (i.e., `SP_scheduling_delay`) is sufficient to cover most setups of the Kafka/Spark cluster.
- Metrics may change their behavior depending on the running application.
  - For the Wordcount application, a Kafka metric (i.e., `KF_fetchconsumer_queue_time`) is sufficient to identify the side where the bottleneck lies as this metric acts as a *reliable* indicator for all setups where WC is the workload, i.e., `setup-0`,`setup-1`, `setup-2`, and `setup-3`.
  - For the Twitter sentiment analysis application, a Spark metric (`SP_scheduling_delay`) is sufficient to identify the side where the bottleneck lies for setups where TSA is the workload i.e., `setup-4`, `setup-5`, and `setup-6`.
  - For the Flight delay prediction application, there is no obvious set of metrics that we can always rely on as *reliable* indicators of performance bottleneck for setups where FDP is the workload, i.e., `setup-7`, `setup-8`, and `setup-9`.
- Figure 3.35 shows a possible metrics combination to cover all setups of the Kafka/Spark use case. It also shows the types of these metrics. Based on Figure 3.35, a combination of three groups of metrics, one metric per group, is sufficient to cover most setups of the Kafka/Spark cluster.
- Figure 3.36 shows a possible combination of server-side-only metrics that cover most setups of the Kafka/Spark use case. Based on Figure 3.36, combining two Kafka metrics with one Spark metric is sufficient to identify performance bottleneck for most setups of the Kafka/Spark use case. Once again we note that these metrics belong to two categories.

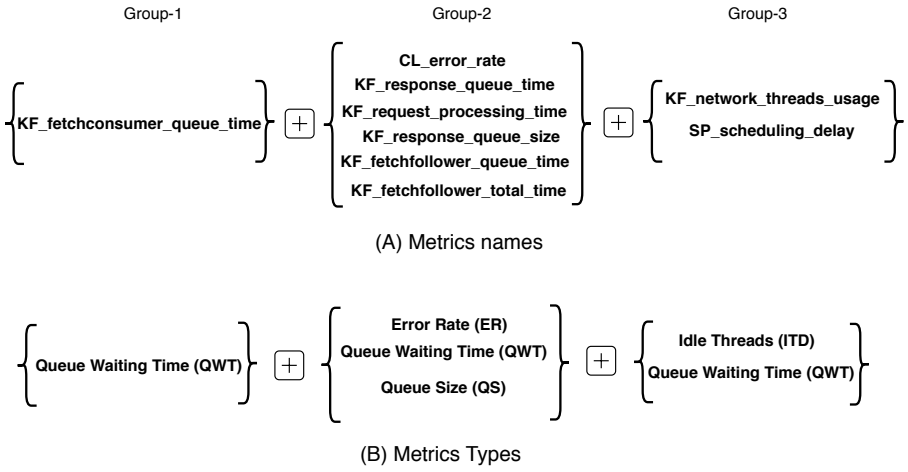


Figure 3.35 – A possible combination of *reliable* indicators for the Kafka/Spark case study.

3.6.5 What are the characteristics of performance bottleneck indicators?

To identify the characteristics of the identified performance bottleneck indicators (i.e., the category these indicators belong to), Figures 3.38, 3.39, and 3.40 show pie charts representing, for each client, Kafka and Spark metrics, respectively, the category that these indicators belong to.

From Figures 3.38, 3.39, and 3.40, we observe that:

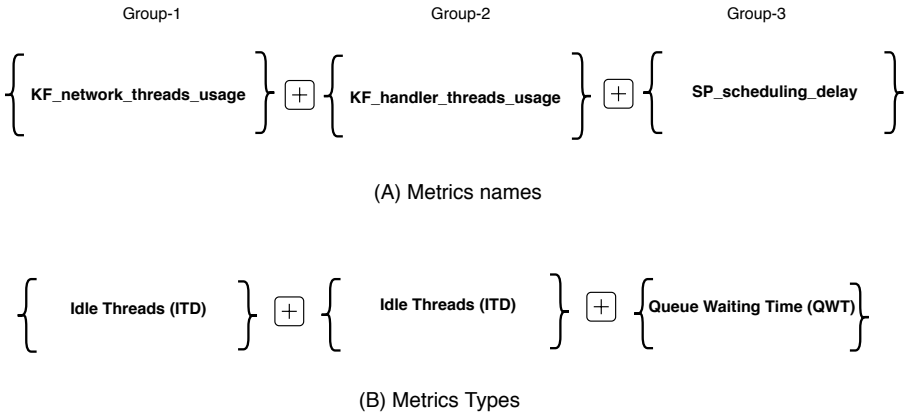


Figure 3.36 – A possible combination of *reliable* server-side-only metrics for the Kafka/Spark case study.

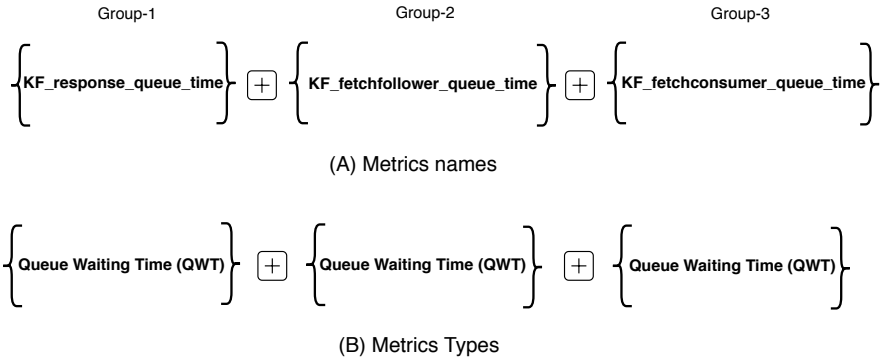


Figure 3.37 – A combination of *reliable* indicators that are never misleading for the Kafka/Spark cluster use case.

Metrics	Setups									
	0	1	2	3	4	5	6	7	8	9
Client metrics										
CL_msg_queue	NR	NR	NR	NR	-	PR	M	PR	PR	PR
CL_error_rate	**R**	NR	NR	PR	-	**R**	NR	M	**R**	**R**
CL_request_latency	NR	NR	NR	PR	-	PR	PR	PR	PR	PR
CL_request_in_flight	M	M	**R**	M	-	PR	M	PR	PR	PR
CL_request_rate	NR	M	M	M	-	NR	NR	NR	NR	NR
CL_response_rate	NR	M	M	M	-	NR	NR	NR	NR	NR
Kafka metrics										
cpu_usage	NR	NR	NR	NR	-	NR	NR	NR	NR	NR
memory_usage	M	NR	NR	NR	-	NR	NR	NR	NR	NR
network_sent	NR	NR	NR	NR	-	NR	M	NR	NR	NR
network_received	NR	NR	NR	NR	-	NR	M	NR	NR	NR
disk_read	NR	NR	NR	NR	-	NR	NR	NR	NR	NR
disk_write	M	NR	NR	NR	-	M	PR	M	NR	NR
KF_network_threads_usage	**R**	M	M	PR	-	PR	**R**	PR	**R**	PR
KF_handler_threads_usage	PR	**R**	**R**	M	-	M	PR	NR	**R**	**R**
KF_response_queue_time	NR	NR	NR	NR	-	**R**	NR	NR	NR	**R**
KF_request_queue_time	NR	NR	NR	M	-	**R**	M	NR	NR	**R**
KF_request_processing_time	NR	M	NR	M	-	**R**	M	NR	NR	**R**
KF_request_queue_size	NR	NR	NR	NR	-	M	PR	PR	PR	PR
KF_response_queue_size	NR	NR	NR	NR	-	**R**	M	PR	**R**	PR
KF_fetchfollower_queue_time	PR	**R**	PR	**R**	-	**R**	PR	PR	**R**	NR
KF_fetchconsumer_queue_time	**R**	**R**	**R**	**R**	-	PR	PR	NR	**R**	**R**
KF_fetchfollower_total_time	NR	NR	NR	NR	-	**R**	M	M	**R**	**R**
KF_fetchconsumer_total_time	NR	NR	NR	NR	-	PR	PR	NR	M	M
KF_producer_purgatorySize	NR	NR	NR	NR	-	NR	NR	NR	NR	NR
KF_fetchfollower_purgatorySize	NR	NR	NR	NR	-	PR	M	M	**R**	**R**
fetchfollower_throttletime	NR	M	M	**R**	-	NR	NR	NR	NR	NR
producer_throttletime	NR	NR	NR	NR	-	NR	M	NR	NR	**R**
Spark metrics										
cpu_usage	NR	NR	NR	NR	-	NR	NR	NR	NR	NR
memory_usage	NR	NR	NR	NR	-	NR	NR	NR	NR	NR
network_sent	NR	NR	NR	NR	-	NR	M	NR	NR	NR
network_received	NR	NR	NR	NR	-	NR	M	NR	NR	NR
disk_read	NR	NR	NR	NR	-	NR	NR	NR	NR	NR
disk_write	NR	NR	NR	NR	-	M	NR	M	NR	NR
SP_scheduling_delay	NR	NR	NR	NR	-	**R**	**R**	M	M	M
SP_processing_time	**R**	PR	NR	M	-	PR	PR	NR	NR	PR
SP_total_delay	NR	NR	NR	NR	-	PR	PR	M	M	M
SP_number_records	NR	PR	PR	PR	-	PR	M	M	PR	PR
KSP_consumer_lag	NR	M	M	NR	-	PR	M	M	PR	NR

Table 3.14 – Analytical study results for all setups of the Kafka/Spark cluster (R: reliable, PR: partially reliable, M: misleading, and NR: not reliable).

- Depending on the tested setup, the category to which a metric belongs may change. Based on the data presented in Table 3.14, we can observe that metrics may change the category they belong to depending on the running application (i.e., Wordcount, Twitter sentiment analysis , or Flight delay prediction).
- Unlike in the Kafka cluster use case, client metrics like `CL_request_latency`, `CL_request_rate`, and `CL_response_rate` are never *reliable* indicators for any given setup of the Kafka/Spark cluster. However, client metrics that belong to the types: **Error Rate** and **Queue Waiting Time** (described in Table 3.2) act as *reliable* indicators for most setups (e.g., `CL_error_rate` and `CL_msg_queue`).
- On the server side, several Kafka metrics (i.e., `KF_handler_threads_usage`, `KF_request_queue_size`, `KF_fetchfollower_queue_time`, and `KF_fetchconsumer_queue_time`) act as *reliable* indicators for most setups. However, none of the Spark metrics act as a *reliable* indicator for most setups (except for Flight delay prediction). In other words, Spark metrics show sensitivity to the tested setup by changing the category to which they belong.

Based on the previous observations, it should be noted that all combinations of metrics presented in Figure 3.35, and 3.36 include at least one metric that becomes misleading on some setups. It means that building a tool that always automatically identifies the bottleneck correctly might be difficult using these metrics. Figure 3.37 shows that selecting 3 server metrics (i.e., `KF_response_queue_time`, `KF_fetchfollower_queue_time`, and `KF_fetchconsumer_queue_time`) is the best combination of metrics including metrics that are never misleading. With these 3 metrics, we can identify the bottleneck in 7 out of 9 cases.

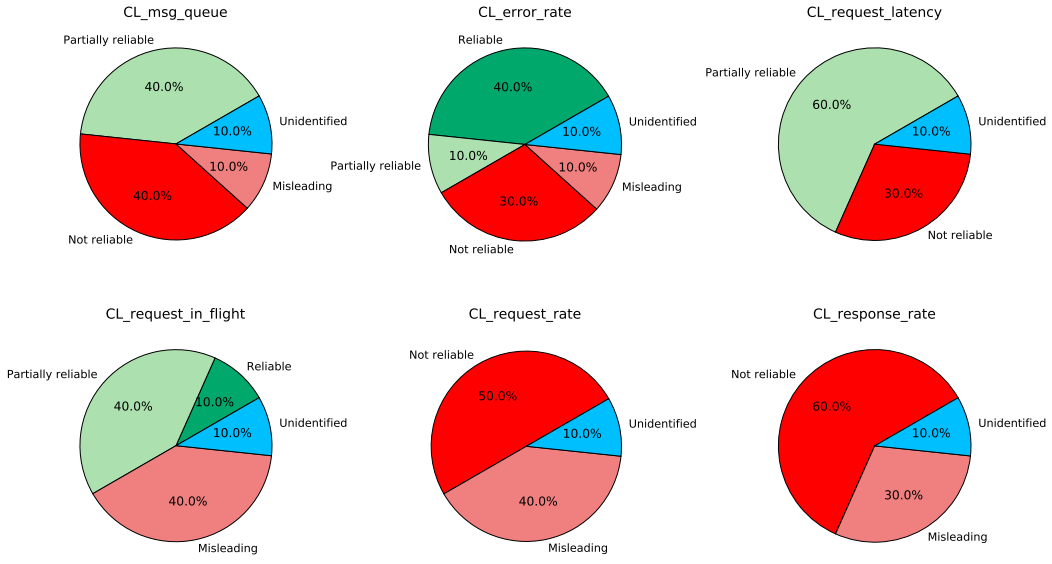


Figure 3.38 – The pattern category (in percent) that client metrics follow for all setups of the Kafka/Spark cluster use case.

3.6.6 How fine-grained are the conclusions that we can draw from these metrics?

In this section, we check whether relying on *reliable* metrics in the Kafka/Spark cluster use case allows us not only to identify if the bottleneck is on the client or on the server side but also, to identify the limiting component in the case of the server being the bottleneck (i.e., Kafka or Spark).

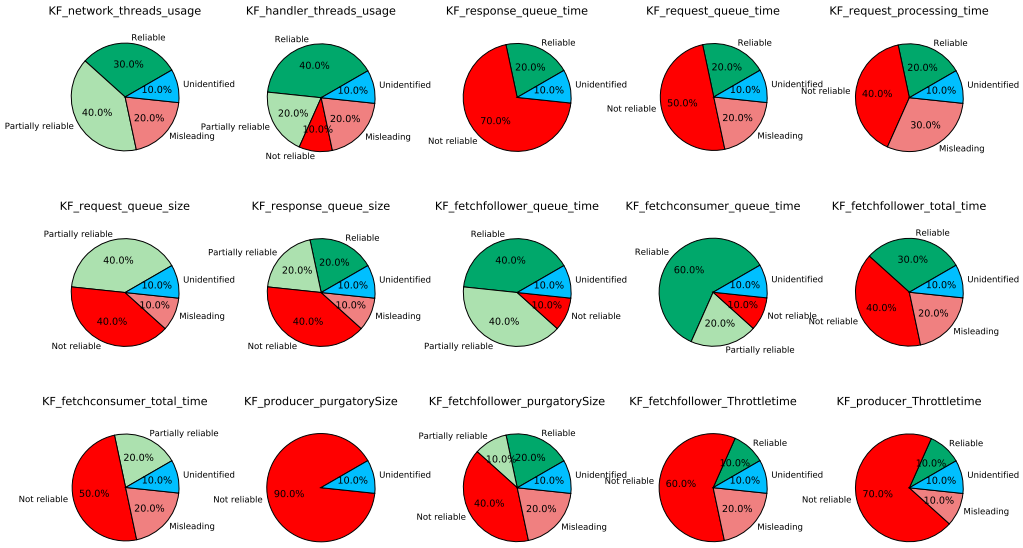


Figure 3.39 – The pattern category (in percent) that Kafka metrics follow for all setups of the Kafka/Spark cluster use case.

To do so, when the bottleneck is on the server side, we tune one server component at a time (Kafka or Spark) by provisioning more nodes. We measure the *global throughput* of the system. If the tuning improves the global throughput of the system, we conclude that the tunned component is the limiting one. A case where we might need to tune the two components on the server side may occur, and in this case, we consider both components as bottlenecks.

To illustrate the described methodology, we consider `setup-0` of the Kafka/Spark cluster (described in Table 3.13). We presented in Figure 3.28 (a) the evolution of the throughput when adding one Kafka or one Spark node. On the figure, we see that with 3 Kafka nodes and 3 Spark nodes, the server reaches its maximum capacity with 18 clients. Figure 3.28 (a) shows that adding one more Kafka server improves the system throughput by up to 43%. However, adding one Spark node does not improve the system throughput. Consequently, we conclude that Kafka is the limiting component in the system.

Now, the question is: Can the *reliable* metrics pinpoint that the Kafka component is limiting the performance in `setup-0`? Figures 3.31 (a) and (i) show evolutions of metrics that belong to the *reliable* category. More precisely, for both metrics, the evolution changes at the point where the bottleneck shifts from the client side to the server side which corresponds to 18 clients. Since both metrics are Kafka metrics, it shows that for this setup, some metrics allow identifying that the Kafka is the bottleneck.
For the other setups, experimentally we find that Kafka is always the limiting component in the Kafka/Spark cluster when the bottleneck is on the server side. In theses cases, some metrics exported by the Kafka component allow identifying that Kafka is the bottleneck. More precisely, based on results presented in Table 3.14, we see that there are several setups where only Kafka metrics act as reliable indicators of performance bottleneck (i.e., `setup-1`, `setup-2`, `setup-3`, `setup-8`, and `setup-9`). In other words, none of the Spark metrics is reliable in those setups. On the other hand, there are setups (i.e., `setup-5` and `setup-6`) where Spark metrics also act reliable indicators (i.e., `SP_scheduling_delay`), and where it would be more difficult to conclude that the Kafka component is the limiting component. Setups where Spark is the limiting component would have to be studied to see if they significantly differ from the setups studied

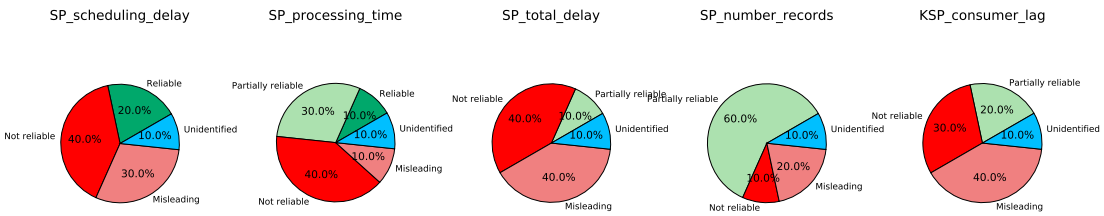


Figure 3.40 – The pattern category (in percent) that Spark metrics follow for all setups of the Kafka/Spark cluster use case.

here.

3.6.7 Discussion

Based on our experimental study on the Kafka/Spark use case with different setups, we summarize our findings as follows:

- Similarly to the Kafka use case, resource consumption metrics (for both Kafka and Spark) are never *reliable* indicators of performance bottlenecks for any given setup of the Kafka/Spark use case.
- Unlike the Kafka use case, client metrics cannot always help in identifying the bottleneck component in all setups of the Kafka/Spark use case.
- Comparing the study findings of the Kafka case and the Kafka/Spark case, metrics may change their behavior depending on the case study. For instance, metrics like `CL_request_rate` and `CL_response_rate` act as *reliable* indicators for most of the tested setups of the Kafka cluster, while they act as *not reliable* or *misleading* indicators for the most setups of the Kafka/Spark case.
- A combination of reliable metrics on both client and server sides can cover almost all studied setups of the Kafka/Spark cluster. For example, combining a client metric (i.e., `CL_error_rate`) with a Kafka metric (i.e., `KF_fetchconsumer_queue_time`) and a Spark metric (i.e., `SP_scheduling_delay`) is sufficient to cover most setups of the Kafka/Spark cluster.
- For each test setup (except for `setup-7`), there are always some metrics that are *reliable* indicators of performance bottleneck. This answers our first question about the existence of *reliable* indicators of performance bottleneck for any given setup of the Kafka/Spark cluster.
- The workload characteristic has an impact on identifying performance bottleneck. For the Wordcount application, one Kafka metric (i.e., `KF_fetchconsumer_queue_time`) is sufficient to identify the side where the bottleneck lies. On the other hand, for the Twitter sentiment analysis application, one Spark metric (i.e., `SP_scheduling_delay`) is sufficient to identify the side where the bottleneck lies. A combination of one client metric, which is `CL_error_rate`, and one Kafka metric, which is `KF_network_threads_usage` can also be sufficient to identify the system bottleneck.

- A combination of server-side-only metrics is sufficient to cover most setups of the Kafka/Spark use case (see Figure 3.36). These metrics belong to two types: **Idle Threads** and **Queue Waiting Time**.
- In general, for the Kafka/Spark case, metrics belonging to these types: **Idle Threads**, **Error Rate**, and **Queue Waiting Time** are *reliable* indicators of performance bottleneck.
- Table 3.15 shows a summary of categories of reliable indicators of performance bottleneck that covers most tested setups of the Kafka/Spark use case.

Metric Type	Client	Server	
		Kafka	Spark
Resource Consumption (RC)			
Idle Threads (ITD)		✓	
Error Rate (ER)	✓		
Queue Waiting Time (QWT)		✓	✓
Queue Size (QS)		✓	
Latency (LY)			
Processing Time (PT)			✓
In/Out Data (IOD)			
Uncategorized (UC)			

Table 3.15 – Summary of reliable metrics types for all setups of the Kafka/Spark cluster use case.

In the next section, we apply the proposed methodology onto the full data processing pipeline where the server is composed of three components: Kafka, Spark Streaming, and Cassandra.

3.7 Case study-3: Multi-tier system (Kafka/Spark/Cassandra cluster)

In this section, we study the full data processing pipeline comprising three main components: Kafka, Spark Streaming, and Cassandra. Similarly, to the previously studied cases, we first illustrate how to apply our approach to one setup of the full data processing pipeline case study. We try to answer the research questions concerning the existence of *reliable* indicators of performance bottleneck and their characteristics. We discuss how fine-grained the conclusions that we can draw from these indicators are (in case of their existence). Finally, we summarize and discuss our findings over all the setups of the Kafka/Spark/Cassandra cluster.

We deploy the data processing pipeline where each of its components has its dedicated nodes i.e., we deploy Kafka instances on 3 dedicated nodes, Spark instances on 3 dedicated nodes, and Cassandra instances on 3 dedicated nodes.

We examine three different applications: Wordcount (WC), Twitter sentiment analysis (TSA), and Flight delay prediction (FDP). We consider 10 different setups of the data processing pipeline described in Table 3.16. For all setups, we consider that the Kafka message size is 100B, the number of Kafka partition is 12, no compression and the acknowledgment setting is one. All setups are performed on Nova cluster (see Table 3.9).

These setups vary depending on the following factors:

- The number of Spark executors: We explained the impact of Spark executors in Section 2.3.2.2. The tested values are: 3 and 12 executors.
- The number of Cassandra nodes: We also vary the number of Cassandra nodes between 1 node and 3 nodes as we want to stress the database component in the system.

- Cassandra consistency level: This setting is defined as the minimum number of Cassandra nodes that must acknowledge a read or write operation before the operation can be considered successful. As the three examined applications request write operations to the database, we only consider the Cassandra consistency level for writing. We vary this value between *1* and *all*. Regarding the read consistency level, we use the default value which is equal to 1.
- Workload type: See Section 3.4.2 for more details.

Setup	Num. Spark executors	Num. Cassandra nodes	Cassandra consistency level	Workload
`setup-0`	3	3	1	WC
`setup-1`	12	3	1	WC
`setup-2`	12	3	all	WC
`setup-3`	3	1	1	WC
`setup-4`	3	3	1	TSA
`setup-5`	12	3	1	TSA
`setup-6`	12	3	all	TSA
`setup-7`	3	3	1	FDP
`setup-8`	12	3	1	FDP
`setup-9`	12	3	all	FDP

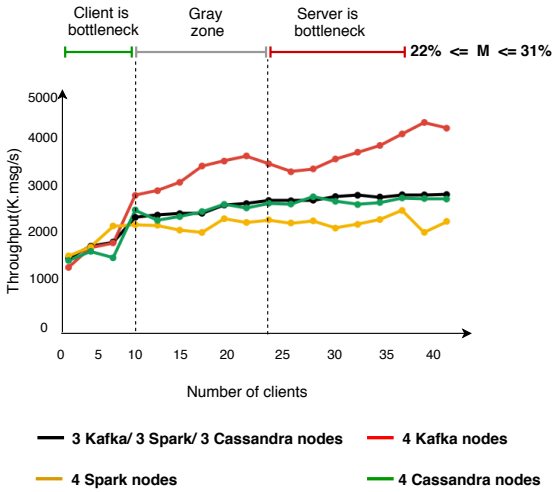
Table 3.16 – Description of the different examined setups for the full data processing pipeline (Kafka/Spark/Cassandra) use case.

3.7.1 Setting the margin value M

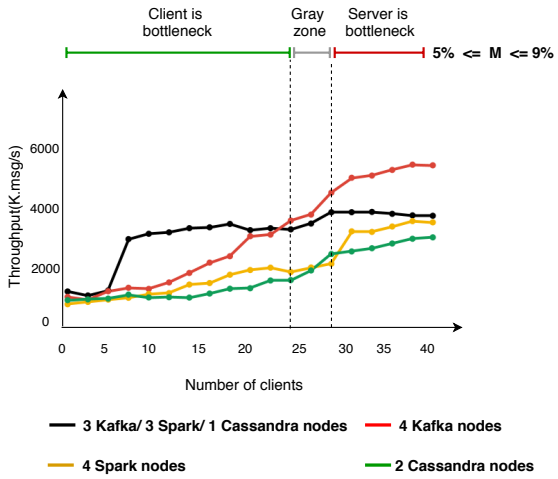
In this section, we apply the same methodology for setting the value of M as described for the Kafka cluster use case (Section 3.5.1). Thus, we only show the output of the methodology on some setups of the full data processing pipeline without presenting the details. The only difference in this use case is that we need to include Cassandra component in the methodology as we have done with Spark instances in the Kafka/Spark use case.

Figure 3.41 shows the system throughput as a function of the number of clients for the number of server nodes (for Kafka, Spark, and Cassandra) equals to 3 and 4 for different setups of the Kafka/Spark/Cassandra cluster. The X-axis represents the number of clients and the Y-axis represents the system throughput in K.msg/s. The black curve represents the system throughput when 3 nodes are used for each component. The red curve represents the system throughput when 4 nodes are used for the Kafka instances in the system (while 3 nodes are used for the Spark instances and 1 or 3 nodes are used for the Cassandra instances depending on the setup). The yellow curve represents the system throughput when 4 nodes are used for Spark instances (while 3 nodes are used for Kafka instances and 1 or 3 nodes are used for Cassandra instances depending on the setup). The green curve represents the system throughput when 2 or 4 nodes are used for Cassandra instances (while 3 nodes are used for Kafka instances and 3 nodes are used for Spark instances).

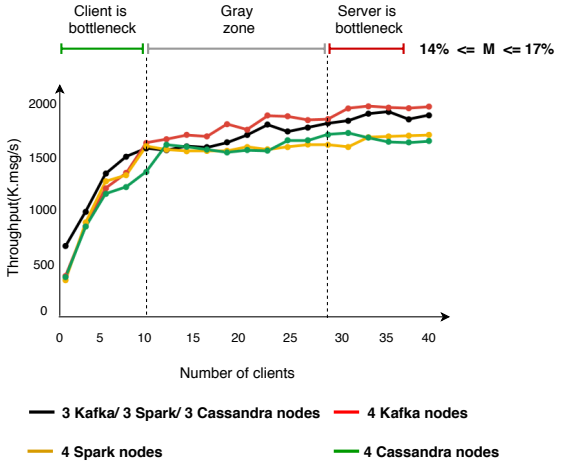
Based on Figure 3.41, we observe that increasing the number of Kafka brokers increases the system throughput at a certain number of clients. However, the ranges of values for M in the different studied setups do not intersect. Since `setup-3` (Figure 3.41 (b)) is the one where wrongly considering the server as the bottleneck could have the most significant impact, we select a value of M that is close to the range for this setup. Hence, we select $M = 10\%$ for the following analysis.



(a) Setup-0.



(b) Setup-3.



(c) Setup-7.

Figure 3.41 – Throughput(K.msg/s) of the full data processing pipeline for different setups and with different number of nodes allocated to each component.

3.7.2 Detailed analysis of one setup

In this section, we consider `setup-0`, defined in Table 3.16, and we explain how we apply the proposed methodology in order to answer the research questions mentioned earlier. We measure the global throughput of the system while increasing the number of clients. Figure 3.42 shows the system throughput as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the system throughput. We compute the *peak throughput range* as following: $[2797 \times 0.95; \quad 2797] = [2657; \quad 2797]$. $Th_{0.95} = 2657$ K.msg/s corresponds to 20 clients. $Th_{limit} = 2391$ K.msg/s corresponds to 10 clients. Based on our definition of the bottleneck, the client is the bottleneck until we reach 10 clients injecting load. Points between 10 clients and 20 clients represent the gray zone. At 20 clients and beyond, the bottleneck shifts to the server side (either to Kafka or to Spark or to Cassandra).

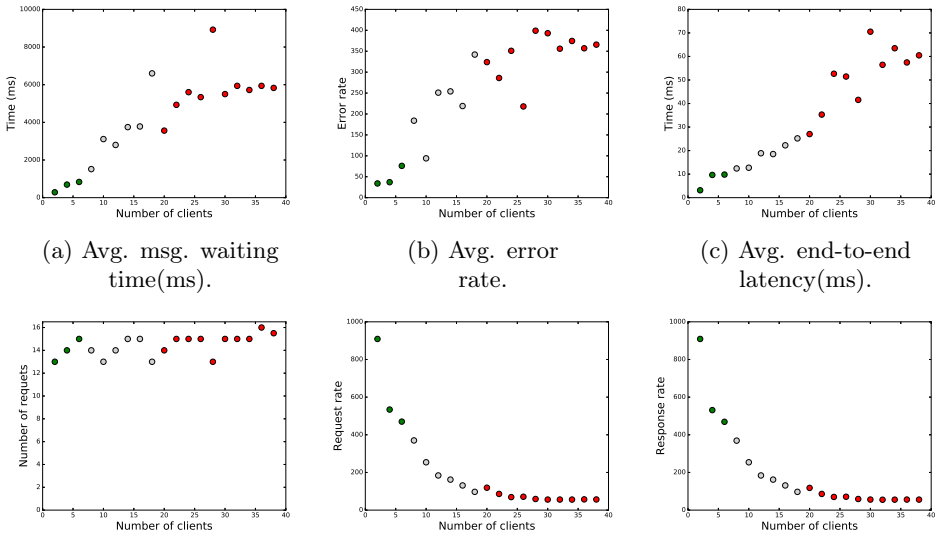
In the following sections, we present the evolution of the metrics for both client and server sides. We analyze the evolution of these metrics. We want to classify them into pattern categories as described in Section 3.3.2.6. The results show that resource consumption metrics always evolve constantly for the tested setup, thus, we do not show the evolution of these metrics.

Figure 3.42 – Bottleneck identification for setup-0 in the full data processing pipeline.

In all the figures presenting the evolution of a metric, the green points represent a bottleneck being on the client side. The gray points represent the gray zone where adding resources to the client or the server side can improve the throughput. The red points represent a bottleneck being on the server side.

3.7.2.1 Evolution of client metrics

Figure 3.43 shows the evolution of the client metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Figures 3.43 (a) and (b) belong to the *reliable* category as there is a change in the values and the behavior at the point where the bottleneck shifts from the client side to the server side. Figures 3.43 (c) and (d) show a constant-linear-periodic variation pattern, thus these metrics belong to the *not reliable* category. Both Figures 3.43 (e) and (f) evolve in a linear-constant way which makes these metrics fall in the *reliable* category.



(a) Avg. msg. waiting time(ms).

(b) Avg. error rate.

(c) Avg. end-to-end latency(ms).

(d) Avg. num. in-flight requests.

(e) Avg. request rate.

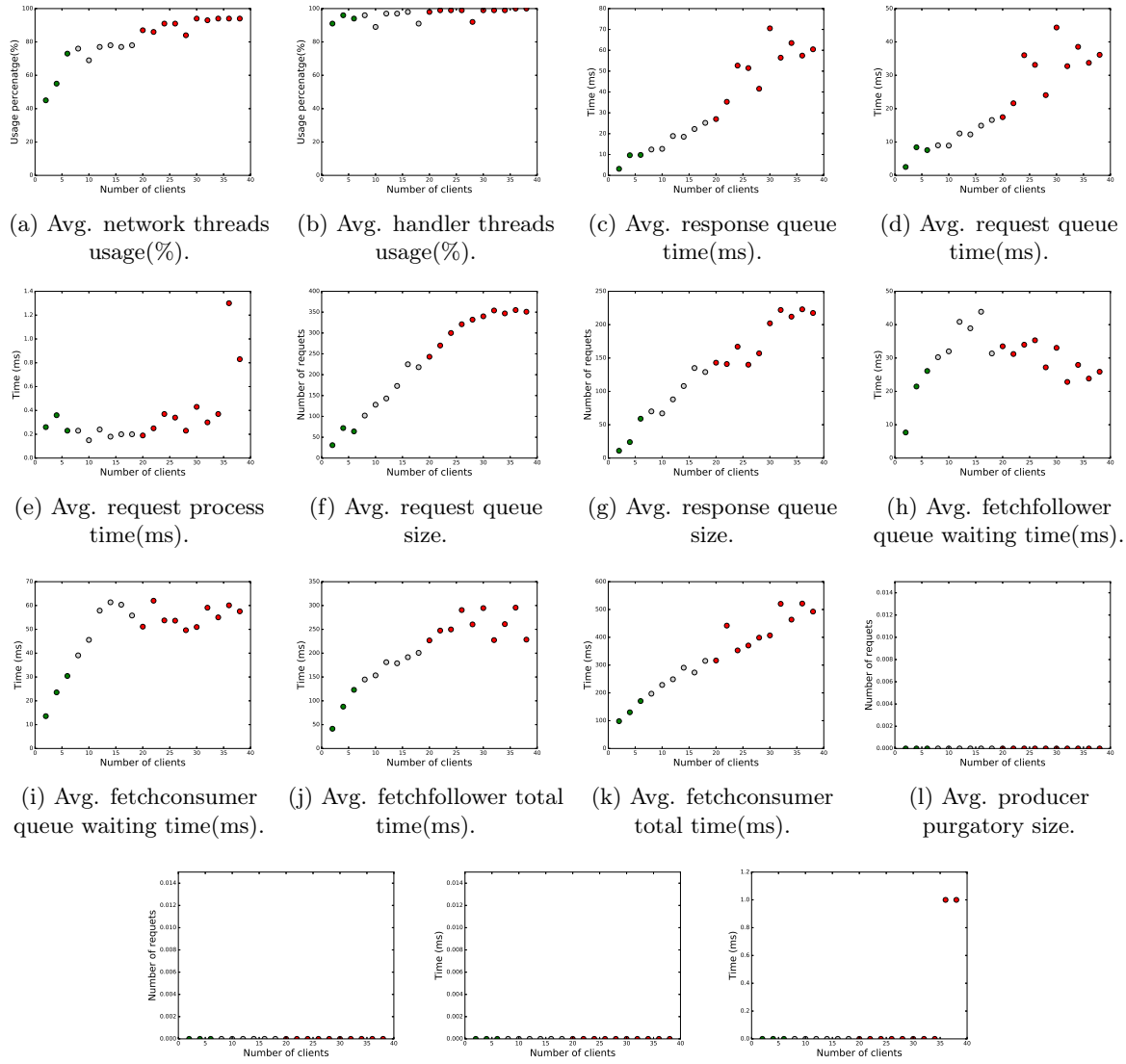
(f) Avg. response rate.

Figure 3.43 – Evolution of client software level metrics for setup-0 in the full data processing pipeline use case.

3.7.2.2 Evolution of Kafka metrics

Figure 3.44 shows the evolution of kafka metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values.

Figure 3.44 shows that `KF_network_threads_usage` (Figure 3.44 (a)) and `KF_fetchconsumer_queue_time` (Figure 3.44 (i)) are the only *reliable* metrics exported by Kafka for `setup-0`. Figure 3.44 (h) evolves in a linear-linear pattern thus it is considered as *partially reliable* indicator. The rest of the metrics are mainly *not reliable* indicators of performance bottleneck. Some metrics are even *misleading* (e.g., Figure 3.44 (o)).



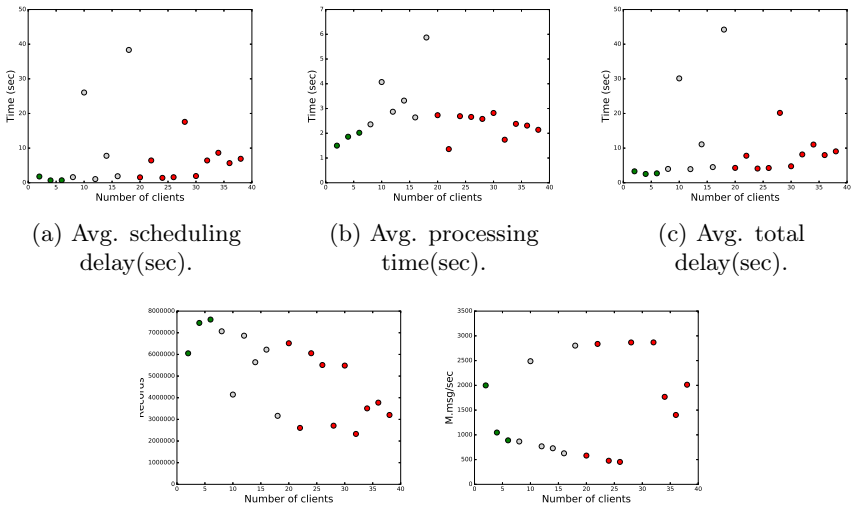
(a) Avg. network threads usage(%).
(b) Avg. handler threads usage(%).
(c) Avg. response queue time(ms).
(d) Avg. request queue time(ms).
(e) Avg. request process time(ms).
(f) Avg. request queue size.
(g) Avg. response queue size.
(h) Avg. fetchfollower queue waiting time(ms).
(i) Avg. fetchconsumer queue waiting time(ms).
(j) Avg. fetchfollower total time(ms).
(k) Avg. fetchconsumer total time(ms).
(l) Avg. producer purgatory size.
(m) Avg. fetchfollower purgatory size.
(n) Avg. fetchfollower throttle time(ms).
(o) Avg. producer throttle time(ms).

Figure 3.44 – Evolution of Kafka software level metrics for setup-0 in the full data processing pipeline use case.

3.7.2.3 Evolution of Spark metrics

Figure 3.45 shows the evolution of Spark metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values.

Based on Figure 3.45, Spark metrics act as either *partially reliable* or *not reliable* indicators of performance bottlenecks. More precisely, the `SP_processing_time` metric (Figure 3.45 (b)) evolves in a linear-periodic way. Thus, this metric belongs to the *partially reliable* category. The rest of the metrics evolve in constant or periodic variation way which makes these metrics belong to the *not reliable* category.



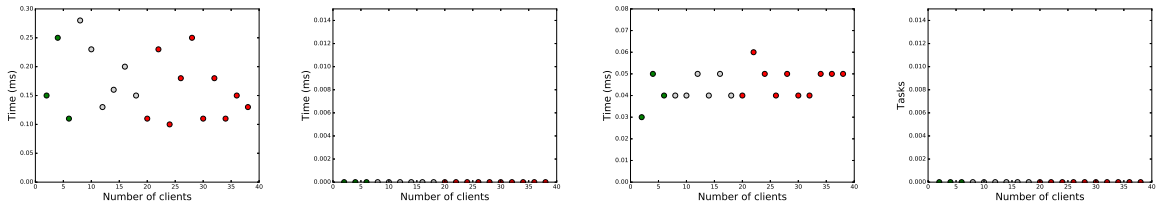
(a) Avg. scheduling delay(sec). (b) Avg. processing time(sec). (c) Avg. total delay(sec).

(d) Avg. number of records. (e) Avg. consumer lag(M.msg).

Figure 3.45 – Evolution of Spark software level metrics for setup-0 in the full data processing pipeline use case.

3.7.2.4 Evolution of Cassandra metrics

Figure 3.46 shows the evolution of Cassandra metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Based on Figure 3.46, we observe that all Cassandra metrics are *not reliable* indicators for the studied setup.



(a) Avg. write latency(ms). (b) Avg. read latency(ms). (c) Avg. req. timeout(ms). (d) Avg. number of pending tasks.

Figure 3.46 – Evolution of Cassandra software level metrics for setup-0 in the full data processing pipeline use case.

In the following sections, we first check if the resource consumption metrics (for Kafka, Spark, and Cassandra) are sufficient for identifying the performance bottleneck for a given setup of the data processing pipeline. Then, we look into the exported metrics by all the system components to verify if there are *reliable*

indicators of performance bottleneck. In the case of the existence of these *reliable* indicators, we try to identify their characteristics. Finally, we summarize and discuss the study findings.

3.7.3 Are resource consumption metrics sufficient?

Similarly to the previous use cases, in this section, we strive to answer the question about resource consumption metrics (i.e., CPU, memory, network, and disk IO) being sufficient indicators of performance bottleneck. We check the resource consumption metrics for all system components: Kafka, Spark, and Cassandra.

Figures 3.47, 3.48, and 3.49 show pie charts representing the categories of resource consumption metrics over all setups of all components of the full data processing pipeline. Based on Figures 3.47, 3.48, and 3.49, we observe that resource consumption metrics are never *reliable* indicators of performance bottlenecks for any tested setup of the Kafka/Spark/Cassandra cluster use case. Hence, we cannot rely on resource consumption metrics as *reliable* indicators of performance bottleneck in the Kafka/Spark/Cassandra cluster use case.

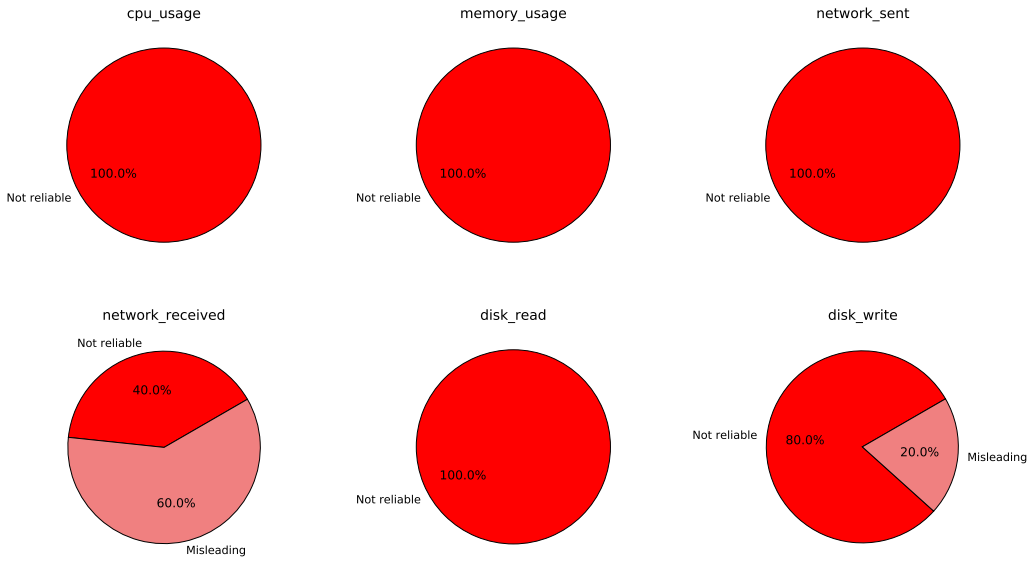


Figure 3.47 – The pattern category (in percent) that Kafka resource consumption metrics follow for all setups of the Kafka/Spark/Cassandra case.

3.7.4 Are there *reliable* indicators of performance bottleneck in a Kafka/Spark/Cassandra cluster?

To answer the question concerning the existence of *reliable* indicators of performance bottleneck for a given setup of the full data processing pipeline, we apply the analytical study on all setups described in Table 3.16. Table 3.17 shows, for each tested setup, the pattern categories of all metrics exported by the Kafka/Spark/Cassandra cluster.

From Table 3.17, we observe that:

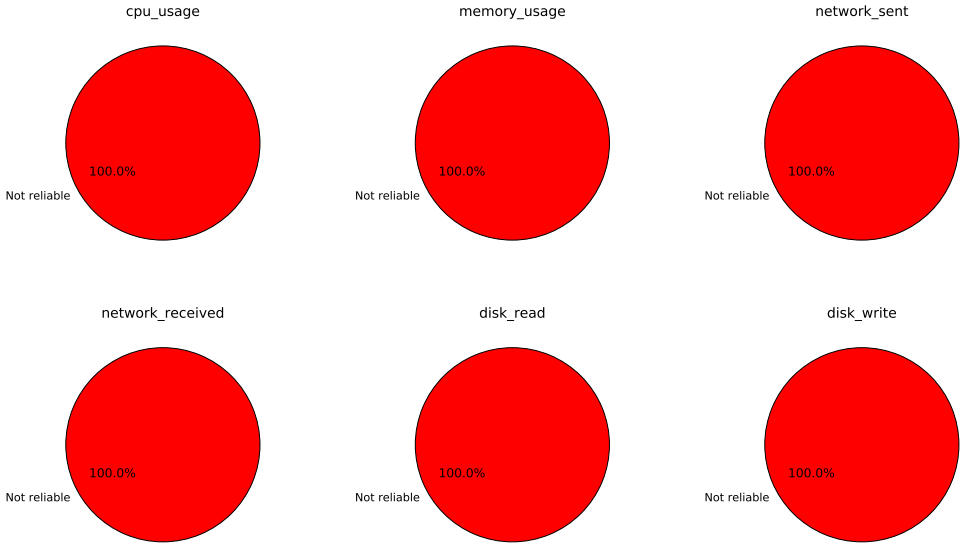


Figure 3.48 – The pattern category (in percent) that Spark resource consumption metrics follow for all setups of the Kafka/Spark/Cassandra case.

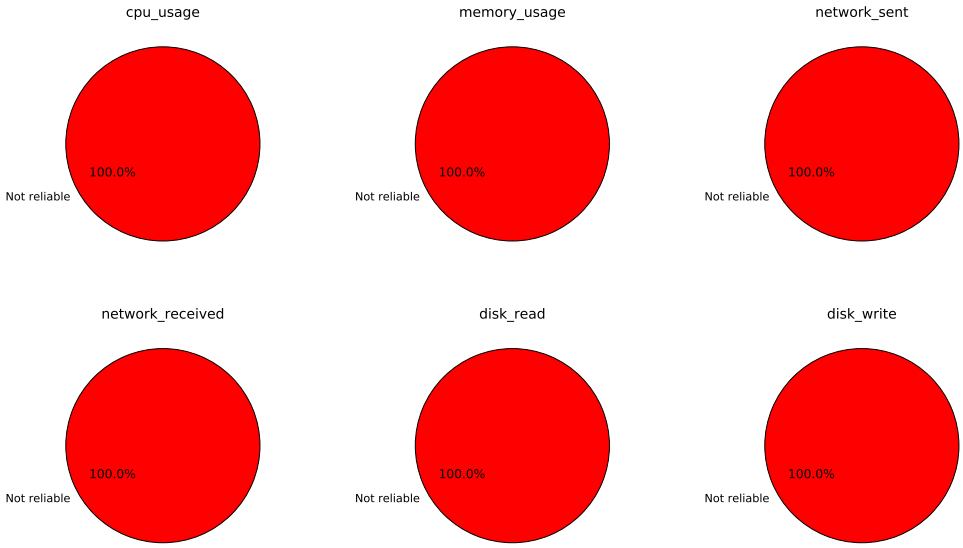


Figure 3.49 – The pattern category (in percent) that Cassandra resource consumption metrics follow for all setups of the Kafka/Spark/Cassandra case.

- For any given setup of the data processing pipeline, there is always at least one *reliable* metric of performance bottleneck.
- For all setups, there is at least one client metric that acts as a *reliable* indicator of performance bottleneck.
- For all setups (except for `setup-4`), there is at least one server metric that acts as a *reliable* indicator of performance bottleneck.
- The Kafka metric `KF_network_threads_usage` participates in identifying the performance bottleneck in most of the setups of the Data processing pipeline, similarly to its behavior for the Kafka cluster case study. Thus, we can conclude that some *reliable* indicators remain valid across very different setups of the data processing pipeline.
- There are always Spark metrics that act as *reliable* or *partially reliable* indicators of performance bottleneck for any given setup.
- Cassandra metrics are never *reliable* indicators for all setups of the data processing pipeline. This is expected as the tested workload applications do not overload Cassandra with read/write requests. Simply put, Wordcount application writes words with their occurrence to the database, Twitter sentiment analysis writes tweets with their sentiment to the database, and Flight delay prediction writes information about flights with the expected delay to the database. These applications do not heavily request Cassandra.
- To cover all the setups of the data processing pipeline, we can combine the Kafka metric (i.e., `KF_network_threads_usage`) with one client metrics (i.e., `CL_request_rate` or `CL_response_rate`) and one Kafka or one Spark metric (i.e., `KF_response_queue_size`, `SP_processing_time`, or `SP_number_records`). Figure 3.50 shows a possible metrics combination to cover all setups of the full data processing pipeline. It also shows the types of these metrics. Based on Figure 3.50, the combination of 3 groups of metrics, one metric per group is sufficient to cover all setups of the data processing pipeline.
- A combination of server-side-only metrics is sufficient to cover most setups of the data processing pipeline. Figure 3.51 shows a possible combination of server side-only metrics to cover 9/10 setups of the data processing pipeline. This combination includes only Kafka metrics.
- Depending on the running workload, the set of reliable metrics that cover all the setups may change.
  - For the Wordcount application (i.e., `setup-0`, `setup-1`, `setup-2`, and `setup-3`), a Kafka metric, `KF_network_threads_usage`, is sufficient to identify the side where the bottleneck lies as this metric acts as a *reliable* indicator for all setups of the Wordcount application. Other possible metrics combinations comprising Kafka metrics, Spark metrics, and Cassandra ones, can be found.
  - For the Twitter sentiment analysis application (i.e., `setup-4`, `setup-5`, and `setup-6`), one of the Kafka metrics, `CL_request_rate` or `CL_response_rate` is sufficient to identify the side where the bottleneck lies.
  - For the Flight delay prediction application (i.e., `setup-7`, `setup-8`, and `setup-9`), the client metric `CL_error_rate` or the Kafka metric `KF_response_queue_size` is sufficient to identify the side where the bottleneck lies. Other possible metrics combinations can be found comprising some client metrics, Kafka metrics, and Spark metrics (no Cassandra metrics).

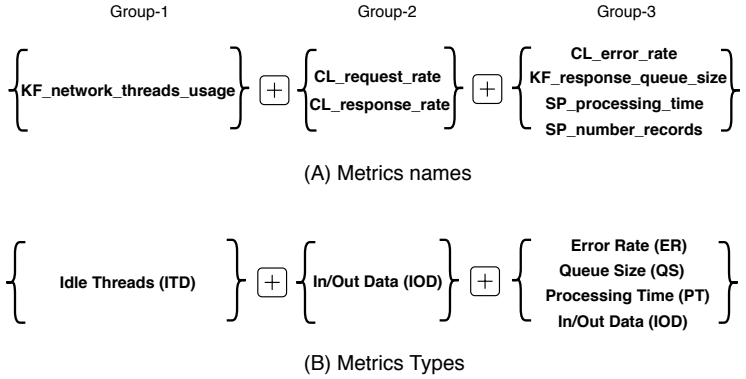


Figure 3.50 – A possible combination of *reliable* indicators for the full data processing pipeline.

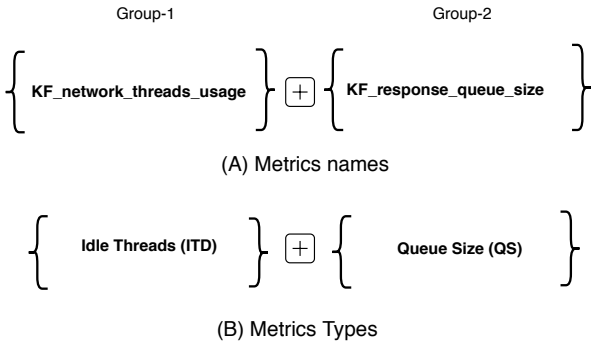


Figure 3.51 – A possible combination of *reliable* server-side-only metrics for the full data processing pipeline.

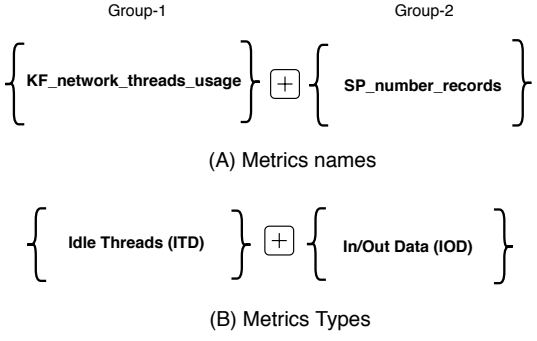


Figure 3.52 – A combination of *reliable* indicators that are never misleading for the full data processing pipeline.

Metrics	Setups									
	0	1	2	3	4	5	6	7	8	9
Client metrics										
CL_msg_queue	R	R	NR	NR	M	R	R	NR	NR	NR
CL_error_rate	R	R	NR	M	PR	R	R	NR	R	R
CL_request_latency	NR	NR	NR	M	M	R	R	NR	R	M
CL_request_in_flight	NR	M	M	M	M	R	R	M	PR	PR
CL_request_rate	R	R	R	PR	R	R	R	NR	NR	NR
CL_response_rate	R	R	R	PR	R	R	R	NR	NR	NR
Kafka metrics										
KF_network_threads_usage	R	R	R	R	NR	R	R	NR	R	R
KF_handler_threads_usage	NR	NR	R	R	NR	R	NR	PR	PR	NR
KF_response_queue_time	NR	NR	NR	NR	M	R	R	M	M	M
KF_request_queue_time	NR	NR	NR	NR	M	R	R	M	M	M
KF_request_processing_time	M	R	NR	R	NR	R	R	M	M	M
KF_request_queue_size	NR	NR	NR	NR	M	R	R	PR	PR	PR
KF_response_queue_size	NR	NR	NR	NR	M	R	R	R	R	R
KF_fetchfollower_queue_time	PR	PR	M	M	PR	R	R	PR	R	R
KF_fetchconsumer_queue_time	R	PR	NR	M	PR	R	PR	NR	NR	M
KF_fetchfollower_total_time	M	PR	PR	NR	M	R	R	NR	M	M
KF_fetchconsumer_total_time	NR	R	NR	NR	PR	PR	M	NR	PR	NR
KF_producer_purgatorySize	M	NR	NR	NR	NR	NR	NR	NR	NR	NR
KF_fetchfollower_purgatorySize	NR	NR	NR	NR	M	R	R	M	M	M
fetchfollower_throttletime	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR
producer_throttletime	M	NR	NR	NR	NR	NR	NR	NR	NR	NR
Spark metrics										
SP_scheduling_delay	NR	PR	NR	R	PR	PR	PR	M	R	R
SP_processing_time	PR	R	PR	NR	M	PR	PR	R	NR	M
SP_total_delay	NR	PR	NR	R	PR	NR	PR	M	R	R
SP_number_records	NR	PR	PR	PR	NR	PR	NR	R	NR	NR
KSP_consumer_lag	NR	R	NR	NR	NR	PR	NR	M	NR	M
Cassandra metrics										
CS_write_request_latency	NR	PR	NR	PR	PR	NR	NR	NR	NR	NR
CS_read_request_latency	NR	NR	NR	NR	NR	NR	NR	NR	M	NR
CS_request_timeout	NR	R	NR	M	M	PR	PR	NR	NR	NR
CS_pending_tasks	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR

Table 3.17 – Analytical study results for all setups of the data processing pipeline (R: reliable, PR: partially reliable, M: misleading, and NR: not reliable).

3.7.5 What are the characteristics of performance bottleneck indicators?

In the previous section, we find that there are some reliable indicators of performance bottleneck in the full data processing pipeline comprising Kafka, Spark Streaming, and Cassandra. In this section, we try to find the characteristics of these indicators. Figures 3.53, 3.54, 3.55, and 3.56 show pie charts that represent, for each client, Kafka, Spark, and Cassandra metric, the category it belongs to respectively.

From Figures 3.53, 3.54, 3.55, and 3.56, we observe that:

- Depending on the tested setup, the category to which a metric belongs may change. Based on the data presented in Table 3.17, we can observe that metrics may change the category they belong to depending on the running application (i.e., Wordcount, Twitter sentiment analysis, or Flight delay prediction).
- On the client side, metrics like `CL_error_rate`, `CL_request_rate`, and `CL_response_rate` act as *reliable* indicators of performance bottleneck for most setups. These metrics belong to the following types (described in Table 3.2): **Error Rate** and **In/Out Data**.
- On the server side, metrics like `KF_network_threads_usage` and `KF_response_queue_size` act as *reliable* indicators of performance bottleneck for most setups. These metrics belong to the following types: **Idle Threads** and **Queue Size**.

Based on the previous observations, it should be noted that all combinations of metrics presented in Figure 3.50, and 3.51 include at least one metric that becomes misleading on some setups. Interestingly, Figure 3.52 shows that combining `KF_network_threads_usage` and `SP_number_records` covers 9 out of 10 setups without ever being misleading.

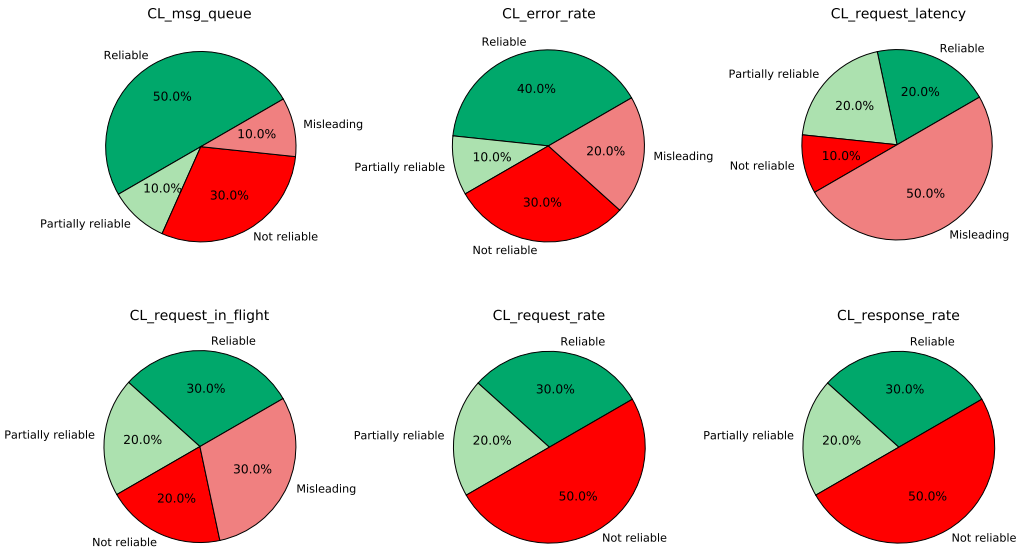


Figure 3.53 – The pattern category (in percent) that client metrics follow for all setups of the full data processing pipeline use case.

3.7.6 How fine-grained are the conclusions that we can draw from these metrics?

In this section, we strive to check if relying on the *reliable* indicators allows us not only identify where the bottleneck is (client or server) but, also, in case the server is the bottleneck, which component is limiting.

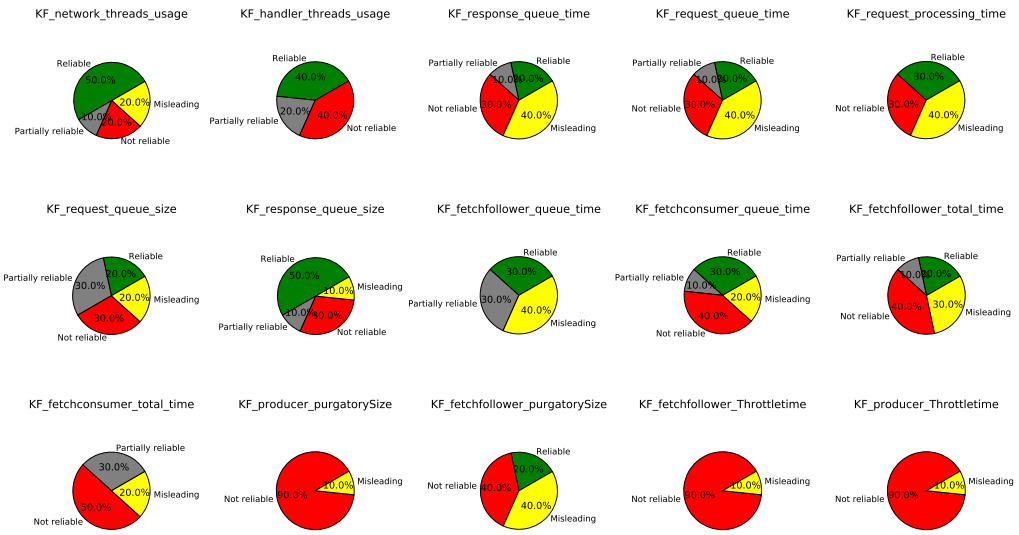


Figure 3.54 – The pattern category (in percent) that Kafka metrics follow for all setups of the full data processing pipeline use case.

To do so, when the bottleneck is on the server side, we tune one server component at a time (Kafka or Spark or Cassandra) by provisioning more nodes. We measure the *global throughput* of the system. If the tuning improves the global throughput of the system, we conclude that the tunned component is the limiting one.

To illustrate the described methodology, we consider `setup-0` of the Kafka/Spark/Cassandra cluster (described in Table 3.16). Figure 3.41 (a) shows the evolution of the throughput when adding one Kafka or one Spark or one Cassandra node. On the figure, we see that with 3 Kafka nodes, 3 Spark nodes, and 3 Cassandra nodes, the server reaches its maximum capacity with 20 clients. Figure 3.41 (a) shows that adding one more Kafka server improves the system throughput by up to 47%. However, neither adding one Spark node nor adding one Cassandra node improves the system throughput. Consequently, we conclude that Kafka is the limiting component in the system.

Now, the question is: Can the *reliable* metrics pinpoint that the Kafka component is limiting the performance in `setup-0`?
Figures 3.44 (a) and (i) show evolutions of metrics that belong to the *reliable* category. More precisely, for both metrics, the evolution changes at the point where the bottleneck shifts from the client side to the server side which corresponds to 18 clients. Since both metrics are Kafka metrics, it shows that for this setup, some metrics allow identifying Kafka as the bottleneck.

For the other setups, based on the results presented in Table 3.17, we see that there are setups where only Kafka metrics act as a reliable indicator of performance bottleneck (i.e., `setup-2`, `setup-3`, `setup-5`, and `setup-6`). On the other hand, there are setups where both Spark and Cassandra metrics also act a reliable indicators, and where it would be more difficult to conclude that the Kafka server is the limiting component. Setups where Spark or Cassandra is the limiting component would have to be studied to see if they significantly differ from the setups studied here.

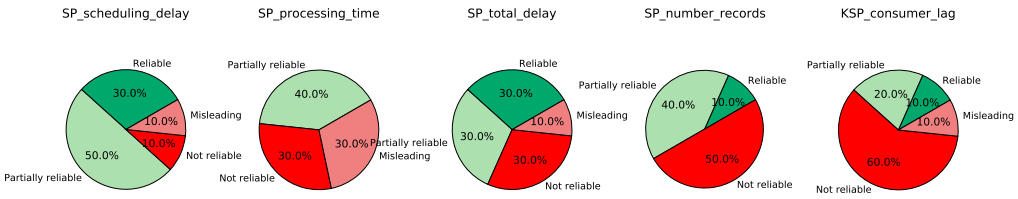


Figure 3.55 – The pattern category (in percent) that Spark metrics follow for all setups of the full data processing pipeline use case.

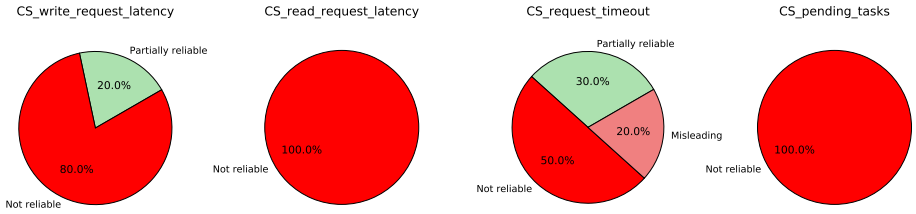


Figure 3.56 – The pattern category (in percent) that Cassandra metrics follow for all setups of the full data processing pipeline use case.

3.7.7 Discussion

Based on our experimental study on the full data processing pipeline, comprising Kafka/Spark/Cassandra, with different setups, we summarize our findings as follows:

- Similar to the other studied cases, resource consumption metrics are never *reliable* indicators of performance bottlenecks for the full data processing pipeline.
- Depending on the running workload, a set of reliable metrics that cover most setups may change.
- There are always *reliable* indicators of performance bottleneck that we can rely on for any given setup of the data processing pipeline. Metrics that belong to the following types: **Idle Threads** (e.g., `KF_network_threads_usage`), **Queue Waiting Time** (e.g., `KF_request_queue_time`) and **Queue Size** (e.g., `KF_response_queue_size`) are *reliable* indicators of performance bottlenecks for most tested cases of the data processing pipeline.
- A combination of server-side-only metrics is sufficient to cover most setups of the data processing pipeline. Figure 3.51 shows a possible combination of server-side-only metrics to cover 9/10 setups of

the data processing pipeline. This combination includes only Kafka metrics.

- Table 3.18 shows a summary of the categories of the reliable indicators of performance bottleneck that covers most tested setups of the data processing pipeline.

Metric Type	Client	Server		
		Kafka	Spark	Cassandra
Resource Consumption (RC)				
Idle Threads (ITD)		✓		
Error Rate (ER)	✓			
Queue Waiting Time (QWT)		✓	✓	
Queue Size (QS)		✓		
Latency (LY)				
Processing Time (PT)			✓	
In/Out Data (IOD)	✓		✓	
Uncategorized (UC)				

Table 3.18 – Summary of reliable metrics types for all setups of the full data processing pipeline use case.

3.8 Summary of the reliable metrics types for the three cases of the data processing pipeline

Throughout the experimental and analytical study that we have performed on different cases of the data processing pipeline, we have obtained the following main results:

- Resource consumption metrics are never *reliable* indicators of performance bottleneck in any setups of the data processing pipeline.
- There are always *reliable* indicators of performance bottleneck for any tested setup of the data processing pipeline[6]. This answers our first question about the existence of *reliable* indicators of performance bottleneck.
- Metrics belonging to the **Latency** type are never *reliable* indicators of performance bottleneck.
- Table 3.20 shows that among the metrics that act as *reliable* indicators, in different studied cases, we always find metrics that belong to the following types: **Idle Threads**, **Queue Waiting Time**, and **Queue Size**.
- There is almost always a combination of server-side-only metrics that can be sufficient to identify the performance bottleneck of a given setup of the data processing pipeline.
- The combinations of metrics that allow to always identify the bottleneck are always combinations of metrics of different types.

[6] With the exclusion of very specific setups where we could not conclude because the same component is always the bottleneck.

Metric Type	Kafka cluster	Kafka/Spark cluster	Full data pipeline
Resource Consumption (RC)			
Idle Threads (ITD)	✓	✓	✓
Error Rate (ER)		✓	✓
Queue Waiting Time (QWT)	✓	✓	✓
Queue Size (QS)	✓	✓	✓
Latency (LY)			
Processing Time (PT)	✓	✓	✓
In/Out Data (IOD)	✓		✓
Uncategorized (UC)			

Table 3.19 – Summary of reliable metrics types for all subsystems of the data processing pipeline.

Metric Type	Kafka cluster	Kafka/Spark cluster	Full data pipeline
Resource Consumption (RC)			
Idle Threads (ITD)	✓	✓	✓
Error Rate (ER)			
Queue Waiting Time (QWT)	✓	✓	✓
Queue Size (QS)	✓		✓
Latency (LY)			
Processing Time (PT)	✓	✓	✓
In/Out Data (IOD)			✓
Uncategorized (UC)			

Table 3.20 – Summary of reliable metrics types for all subsystems of the data processing pipeline.

3.9 Conclusion

In this chapter, we have presented our analytical study to identify reliable indicators of performance bottlenecks in the data processing pipeline. We studied different subsystems of the data processing pipeline including the Kafka cluster use case, the Kafka/Spark cluster use case, and the full pipeline comprising Kafka, Spark, and Cassandra. For each system, we considered different setups (i.e., 35 setups in total).

We tested three different workload applications that vary in their characteristics between applying simple operations on the input data (i.e., Wordcount) to using machine learning techniques (i.e., Flight delay prediction).

For each studied system:

- We show that the intuitive approach of monitoring the system-level metrics (i.e., CPU, memory, network, and disk IO) to identify performance bottleneck in distributed systems is *NOT* sufficient as experimentally we show that resource consumption metrics are never *reliable* indicators of performance bottleneck in any of the tested setups of the data processing pipeline.

- We show that among the hundreds of metrics exported by the components of the data processing pipeline, there are always *reliable* indicators of performance bottleneck that can be used to identify the performance bottlenecks in the system. This answers the question concerning the existence of *reliable* indicators of performance bottleneck.

- We also show how to infer these *reliable* indicators and what their characteristics are. Moreover, we show that metrics that belong to these metrics types: **Idle Threads**, **Queue Waiting Time**, **Queue Size**, and **Processing Time** (see Table 3.20) are *reliable* indicators of performance bottleneck of the studied

systems. On the other hand, metrics that belong to **Latency** type, are never *reliable* indicators for the studied systems. This answers the question concerning the characteristics of the *reliable* indicators of the performance bottleneck.

- We observe that when relying only on never misleading metrics, it becomes difficult to find a combination of metrics that can always act as *reliable* indicators for all setups.

This last comment can make think that it is difficult to build a tool that automatically identifies where the bottleneck is in the data processing pipeline. To get more inputs, one could also take into account some metrics we considered as non-reliable because they require setting a threshold to identify the point where the server reaches its maximum capacity.

In Chapter 4, we show that using machine learning techniques can help in analyzing exported metrics to identify the limiting component in the data processing pipeline. Furthermore, we show that the selection of reliable metrics based on the criteria presented in this chapter, can help creating more accurate machine learning models.

Chapter 4

A Learning-Based Approach for Performance Bottlenecks Identification

Contents

In this chapter, we study the use of machine learning techniques to build models that can automatically identify performance bottlenecks in the data processing pipeline. First, we describe the context and the motivation of using a learning-based approach for troubleshooting performance bottlenecks in the data processing pipelines in Section 4.1. We also present in detail the methodology we adopt for building and evaluating learning models. We discuss the related research works that have used machine learning techniques for performance identification in distributed systems in Section 4.2. Finally, in Section 4.3, we run evaluations through different use cases of the data processing pipeline including: a Kafka cluster, a Kafka/Spark cluster, and a Kafka/Spark/Cassandra cluster. We examine different machine learning classifiers including Decision Tree (DT) [80], Random Forest (RF) [157], Support Vector Machines (SVM) [65], and Logistic Regression (LR) [77].

We evaluate models built out different subsets of metrics and compare their accuracy. The considered subsets of metrics include subsets used in some related works [108,121,148,149,161] as well as metrics selected based on our work in Chapter 3.

Finally, we discuss and summarize the findings of the learning-based approach in Section 4.4.

4.1 Motivation and background

In this section, we present the motivation and the goal of using a learning-based approach for performance bottleneck identification in the data processing pipeline described in Chapter 2. Then, we give the necessary background related to machine learning techniques we use to identify performance bottlenecks in the studied cases.

4.1.1 Motivation

Today's systems generate a huge quantity of health and operational data that can easily overwhelm the analysis and detection processes. In addition, it is a tedious task to be done by humans. Finding bottleneck symptoms and limiting components in such datasets is similar to finding a needle in a haystack.

Many exiting works show efficiency in applying machine learning techniques to different problem domains [81, 85, 93, 158]. In the context of performance studies, machine learning techniques have been used to troubleshooting performance issues in distributed systems [106, 109, 158].
We have seen in Chapter 3 that relying on *reliable* indicators can help to identify on what side of the data processing pipeline (client or server) the bottleneck lies. On the other hand, the results of Chapter 3 show that when changing the setup, it is not the same metrics that will behave as reliable indicators. Furthermore, a metric that is a reliable indicator in some setups may become misleading in other setups. Hence, evaluating whether the server has reached its maximum capacity in a new configuration of the system can still be challenging. In this chapter, we show how machine learning can be used to build models that analyze metrics to automatically determine whether the data processing pipeline has reached its peak throughput in a given configuration.

4.1.2 Goal

Using machine learning techniques, we aim at :

- Building a tool that determines whether the server has reached its maximum capacity in the data processing pipeline. The tool uses a set of metrics exported by system components on both hardware and software levels.
- Identifying a set of metrics that allows building learning models that achieve high prediction accuracy for performance bottleneck identification.
- Quantifying the robustness of the obtained models when generalizing to new setups, to new numbers of clients, or to both new setups and new numbers of clients.

4.1.3 Machine learning techniques: A background

Machine learning algorithms can be classified into two broad categories based on the nature of the input and the expected output of the algorithms: Supervised Learning and Unsupervised Learning [66]. In this thesis, we use Supervised Learning since we are working on labeled data. Thus in the following we give a brief background related to Supervised Learning, focusing on classification algorithms. It worth to mention that both neural networks and deep learning are not considered in our study due to the limited number of samples. Moreover, the results of these techniques (i.e., neural networks and deep learning) are difficult to interpret compared with the results from the considered classifiers.

4.1.3.1 Supervised learning and classification

Supervised learning learns a function that maps an input to an output based on example input-output pairs [136]. It infers a function from labeled training data consisting of a set of training examples [124]. A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples.

Classification corresponds to the set of supervised learning techniques that solve the problem of classifying a new observation in a set of categories (classes) [119]. Many domains apply machine learning techniques in their applications (e.g., credit approval, medical diagnosis, target marketing, etc.). There are several classification algorithms, and choosing which algorithm to apply depends on the application and nature of the available data set.

In this chapter, we examine models with 4 common classifiers: Decision Tree (DT), Random Forest(RF), Support Vector Machines (SVM), and Logistic Regression (LR). In the next sections, we describe each classifier briefly.

4.1.3.2 Learning decision trees

Decision trees create a hierarchal partitioning of the data, which relates different partitions at the leaf level to the different classes. The hierarchal partitioning at each level is created with the use of a *split criterion*. The split criterion may either use a condition on a single attribute, or it may contain a condition on multiple attributes [65]. The goal of learning decision trees is to create a model that predicts the value of a target variable based on several input variables. In a decision tree or a classification tree, each internal (non-leaf) node is labeled with an input feature. The arcs coming from a node labeled with an input feature are labeled with each of the possible values of the target, or the output feature, or the arc leads to a subordinate decision node on a different input feature. Each leaf of the tree is labeled with a class or a probability distribution over the classes, signifying that the data set has been classified by the tree into either a specific class, or into a particular probability distribution (which, if the decision tree is well-constructed, is skewed towards certain subsets of classes). Figure 4.1 shows a simple decision tree for classifying when the bottleneck is on the client side and when it is on the server side[1]. In other words, for this example we have two classes: `client` and `server`. We use decision trees as they show the crucial advantage of yielding human-interpretable results, which is important if the method is to be adopted by real systems operators [80].

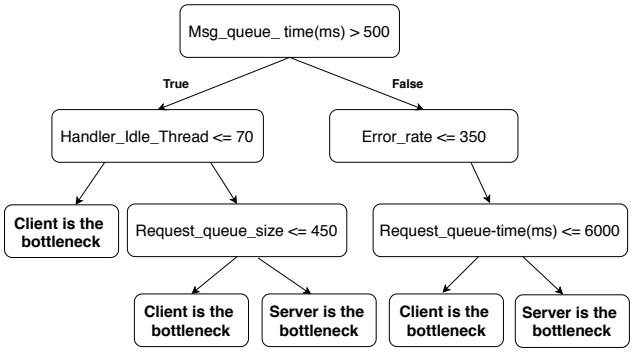


Figure 4.1 – Illustration of a decision tree.

[1]This example is presented only for illustrative purpose. It does not correspond to a tree generated during our study.

4.1.3.3 Random forests

Random forests [157] or random decision forests are an ensemble learning method for classification, regression, and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees habit of overfitting to their training set. The fundamental concept behind random forests is a simple but powerful one — the wisdom of crowds. In the data science domain, the reason that the random forest model works so well is: A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models. The low correlation between models is the key. Just like how investments with low correlations come together to form a portfolio that is greater than the sum of its parts, uncorrelated models can produce ensemble predictions that are more accurate than any of the individual predictions. The reason for this effect is that the trees protect each other from their errors (as long as they do not constantly make errors in the same direction). While some trees may be wrong, many other trees will be right, so as a group, the trees can move in the correct direction. Moreover, in a decision tree, when it is time to split a node, we consider every possible feature and pick the one that produces the most separation between the observations in the left node vs. those in the right node. In contrast, each tree in a random forest can pick only from a random subset of features. This forces even more variation amongst the trees in the model and ultimately results in lower correlation across trees and more diversification. Figure 4.2 shows an example of node splitting in a random forest model and a decision tree model. Figure 4.2 shows that the traditional decision tree (in blue) can select from all four features when deciding how to split the node. It decides to go with Feature 1 as it splits the data into groups that are as separated as possible. On the same figure, we see two of the forest's trees where random forest Tree 1 can only consider Features 2 and 3 (selected randomly) for its node splitting decision. We know from our traditional decision tree (in blue) that Feature 1 is the best feature for splitting, but Tree 1 cannot see Feature 1 so it is forced to go with Feature 2 . Tree 2, on the other hand, can only see Features 1 and 3 so it is able to pick Feature 1. So in the random forest algorithm, we end up with trees that are not only trained on different sets of data but also use different features to make decisions.

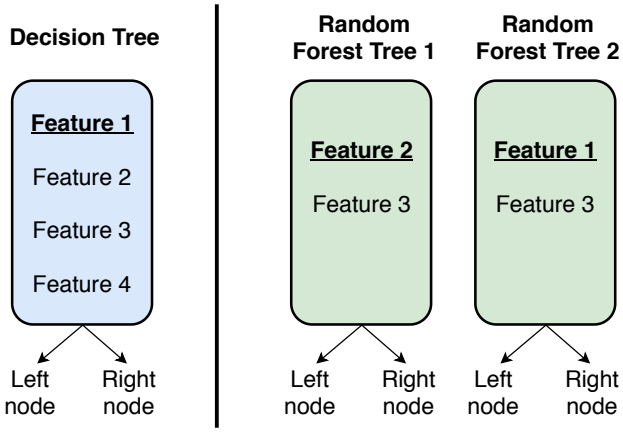


Figure 4.2 – Node splitting in a random forest model is based on a random subset of features for each tree [57].

4.1.3.4 Support vector machines

Support vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of more categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on the side of the gap on which they fall [90].

In addition to performing linear classification [65], SVMs can efficiently perform a non-linear classification, implicitly mapping their inputs into high-dimensional feature spaces.

4.1.3.5 Logistic regression

Logistic regression [77] is a statistical method for analyzing a data set in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are two possible outcomes or more). The goal of logistic regression is to find the best fitting model to describe the relationship between the dichotomous characteristic of interest and a set of independent (predictor or explanatory) variables. This is better than other binary classification algorithms like nearest neighbor since it also explains quantitatively the factors that lead to classification. Logistic regression is one of the most simple and commonly used machine learning algorithms for two-class classification [58]. It is easy to implement and can be used as the baseline for any binary classification problem. Its basic fundamental concepts are also constructive in deep learning. Logistic regression describes and estimates the relationship between one dependent binary variable and independent variables. It is a special case of linear regression where the target variable is categorical in nature. It uses a log of odds as the dependent variable. There are different types of logistic regression: (i) Binary where the target variable has only two possible outcomes such as Spam or Not Spam; (ii) Multinomial where the target variable has three or more nominal categories such as predicting the limiting component in a multi-tier system; (iii) Ordinal where the target variable has three or more ordinal categories such as restaurant or product rating from 1 to 5.

4.2 Related work

There has been much research conducted to benefit from machine learning techniques to improve the performance of systems in different areas [81, 85, 93, 106, 109, 158].

Some of these works have used learning-based approaches to predict the sensitivity of multi-threaded applications to the placement policies for its threads and try to identify the best placement policy in NUMA architectures [93]. Castro et al. [81] propose a machine-learning-based approach to automatically infer a suitable thread mapping strategy for transactional memory applications. Zhang et al. [159] present two approaches based on deep neural networks (DNNs) to reduce the GPU memory consumption efficiently and effectively in deep learning applications.

In the database domain, reinforcement learning has been used to find optimal configurations for cloud database systems for tuning and achieving higher performance [158]. CDBTune [158] is an end-to-end automatic cloud database tuning system that uses deep reinforcement learning. CDBTune utilizes the deep deterministic policy gradient method to find the optimal configurations. CDBTune adopts a try-and-error method to enable utilizing a few samples to achieve higher performance (high throughput and low latency).

In their work, the authors focus on metrics that indicate high-resource utilization. Their analysis tool utilizes simple machine learning techniques to classify the resource consumption metrics and find potential bottlenecks. In our case, we consider all kinds of metrics on both hardware and software levels. We show in Chapter 3 that resource consumption metrics are not good indicators in pinpointing the bottleneck component in the studied systems. Furthermore, we will show in the evaluation section that models based on the resource consumption metrics achieve low predication accuracy of performance bottlenecks in the data processing pipeline.

Moreover, machine learning techniques have been introduced to troubleshooting and identifying performance issues in systems [106, 109, 131, 158].

Parekh et al. [131] use machine learning classifiers to identify bottlenecks in the hardware and software configuration of multi-tier applications. The main idea is to use machine learning classifiers with many metrics to identify particular bottleneck metrics that indicate application misconfiguration leading to failed service-level objectives (SLOs). The main purpose is to improve the SLOs of applications during operations. In their experiments, they initially collect 220 application-specific and system-level metrics. Then they compare metrics to the overall response time. From that function, they apply a threshold to reduce their working set of metrics for training. They explore the performance of three machine learning classifiers (tree-augmented Naïve Bayesian network, a J48 decision tree, and LogitBoost) concerning bottleneck detection in enterprises, multi-tier applications governed by service-level objectives. Their results show the efficiency of machine learning techniques to solve such a problem. However, our problem differs from theirs since we are not governed by some service-level objectives. We aim at finding reliable metrics of performance bottleneck without filtering metrics based on specified thresholds to satisfy some declared objectives.

Jung et al. [109] use machine learning to determine service-level objectives satisfaction and locate bottlenecks in candidate deployment scenarios of distributed multi-tiered applications. The main idea is to verify and validate the configuration performance by a pre-production process referred to as staging. They are interested in metrics that strongly correlate with SLOs violations. They try to answer two questions (i) Does the application configuration meet its performance requirements?; (ii) If the requirements are not met, then is that configuration the bottleneck? To answer the first question, for each trial, their SLO-evaluator uses exported data and computes application-specific throughput and average response time. The second question is answered with aggregated data from multiple trials. If SLO is not met, the tools begin a three-step bottleneck detection process to correlate performance shortfalls and metrics. They first start with the low workload, and each subsequent trial increments the workload until consecutive trials fail the SLO. Mainly, they identify the bottleneck tier by computing the duration of the request in each tier, and the fastest growing in the duration defines the bottleneck tier. Then for the bottleneck tier, they correlate the change in metrics values (they set a threshold of this change to be considered) to one category of SLOs satisfaction (they define 5 categories: 0%, 25%, 50%, 75%, and 100%). This work differs from our problem because their goal is to satisfy service-level objectives (i.e., the duration of the request) by correlating metrics with SLO violations. Moreover, a change in metrics values is detected based on predefined thresholds. However, in our work, we are not interested in satisfying some objective functions. Also, we study the evolution of metrics without setting any threshold to detect the change in their values as we believe these thresholds would need to be tuned when examining different setups of the studied system, which would be a tedious task.

Cohen et al. [85, 87] apply a tree-augmented Naïve Bayesian network to discover correlations between system-level metrics and performance states, such as SLO satisfaction and SLO violation. They also present a decision tree learning approach to diagnose failures in large Internet sites. The main idea is to record runtime properties of each request and apply automated machine learning and data mining techniques to identify the

causes of failures. Their approach analyzes readily available request traces (request logs) to automatically locate sources of error. The approach relies on metrics that are highly correlated with failures. Again, this work differs from ours because they rely on identifying combinations of system-level metrics and threshold values that correlate with high-level performance states. The work by Bodik et al. [78] is another example of a contribution that uses machine learning to evaluate the performance of distributed applications. In this case, machine learning is used to control the number of servers allocated to cloud applications. Once again, the system relies on SLO violations to make predictions. Powers et al. [133] use data mining and machine learning techniques to predict upcoming periods of high utilization or poor performance in enterprise systems. Their goal is to automate the assignment of resources to stabilize performance (e.g., adding servers to a cluster), or to apply opportunistic job scheduling (e.g., backups or virus scans). In other words, rather than detecting bottlenecks in the current system, they predict whether the system will be able to withstand load in the following hour. Using machine learning methods, they aim to predict whether the number of SLO violations in the next hour will exceed a specified threshold.

Seer [100] is an online cloud performance debugging system that leverages deep learning and the massive amount of tracing data cloud systems collect to learn spatial and temporal patterns that translate to SLO violations. Seer combines lightweight distributed RPC-level tracing, with detailed low-level hardware monitoring to signal an upcoming SLO violation, and diagnose the source of unpredictable performance. Once an imminent SLO violation is detected, Seer notifies the cluster manager to take action to avoid performance degradation altogether. Seer focuses on predicting SLO violations in a running environment with a few hundreds of milliseconds of lead time. Hence, their results are not directly applicable to our use case.

Finally, Rao et al. [134] apply machine learning techniques using hardware performance counters to determine, similarly to us, when a server has reached its maximum capacity. Their study focuses on a Web stack, and contrary to us, they assume that the workload is known in advance. Furthermore, our evaluation in Chapter 3 shows that hardware metrics are most often not good indicators of performance bottlenecks in the data processing pipeline.

The study of the related work shows that machine learning techniques have already been used successfully to solve different performance issues in distributed systems, and especially in multi-tier systems. We note, however, that most of the works focus on predicting when a distributed service is going to violate some service-level objectives. Relying on such a service-level objective to determine when an alarm should be raised is a different problem from the one we are trying to tackle from two points of view. On one hand, it is a simpler problem because it allows drawing conclusions from high-level metrics whereas in our case, it is difficult to determine in advance what is the maximum achievable throughput in a given setup of our data processing pipeline. On the other hand, it can be a more difficult problem because the system might violate a service-level objective before reaching its maximum capacity. The work by Rao et al. [134] shares the same objective as us. However, it relies on information about the expected load to make predictions, we think that this information can sometimes be difficult to obtain in practice.

Regarding the data used to build machine learning models for predictions, we observe that a large number of works rely on low-level hardware metrics. Our study, presented in Chapter 3, shows that hardware metrics are often of little help in the case of our data processing pipeline. To understand which metrics can provide the best results for our case study, during our evaluation we compare models built based on different sets of metrics. This comparison shows that models based on hardware metrics achieve a low prediction accuracy.

4.3 Building and evaluating models to identify performance bottlenecks

This section studies how to build a model that can accurately identify performance bottlenecks in the data processing pipeline. More precisely, we study what metrics should be given as input to a classification algorithm to be able to accurately identify a performance bottleneck. To this end, we compare the accuracy obtained when building models using different subsets of metrics including subsets that correspond to metrics used by some related works [109, 150, 161]. The sets of metrics we consider are:

- **All**: It represents models that use *all* metrics exported by the system components (i.e., client and server) on both hardware and software levels.
- **Recommended**: It represents models that use only *recommended* metrics. By recommended metrics, we refer to metrics identified as reliable in Chapter 3. For the Kafka cluster use case, the set of *recommended* metrics includes: `CL_request_rate`, `CL_response_rate`, `CL_error_rate`, `CL_request_in_flight`, `KF_handler_threads_usage`, `KF_network_threads_usage`, `KF_request_queue_time`, `KF_response_queue_time`, `KF_request_processing_time`, `KF_request_queue_size` and `KF_fetchfollower_queue_time`. For the Kafka/Spark cluster use case, the set of *recommended* metrics includes the set of recommended metrics for the Kafka cluster listed just above plus the following metrics: `KF_fetchconsumer_queue_time`, `KF_fetchfollower_total_time`, `KF_response_queue_size`, and `SP_scheduling_delay`. For the full data processing pipeline use case, the set of *recommended* metrics includes: `CL_request_rate`, `CL_response_rate`, `CL_error_rate`, `KF_handler_threads_usage`, `KF_network_threads_usage`, `KF_response_queue_size`, `SP_processing_time`, and `SP_number_records`.
- **Server-side-only**: It represents models that use only metrics exported on the software level (resource consumption metrics are excluded in these models) by components on the server side, i.e., we do not include client metrics in these models.
- **Resource consumption (RC)**: It represents models that use only resource consumption metrics (i.e., CPU, memory, network, and IO).
- **Response time (RT)**: It represents models that use only response time metrics, as described in Chapter 3.
- **Waiting time (WT)**: It represents models that use only waiting time metrics, as described in Chapter 3.

We evaluate these models with 4 different classifiers including Decision Tree (DT) [80], Random Forest (RF) [79], Support Vector machine (SVM) [65], and Logistic Regression (LR) [77]. We consider multiple classifiers to ensure that our conclusions are not simply due to a bad choice of a classifier. We use the classifiers as black boxes: we use the default classification functions without tuning their hyperparameters [146]. We chose to use classifiers as black boxes because our goal is to demonstrate the feasibility of the approach and not to find the most optimal classification algorithm for our problem. Furthermore, using default classification functions shows that one can apply our methodology without having a high degree of expertise in machine learning.

We consider the three subsystems of the data processing pipeline described in Chapter 3. For each use case

(i.e., a Kafka cluster, a Kafka/Spark cluster, a Kafka/Spark/Cassandra cluster), we strive to answer the following questions:

- Can we rely on machine learning techniques applied to monitored metrics to identify on what side the bottleneck lies (i.e., client or server)?
- What kind(s) of metrics should we rely on to build an accurate learning model?
- Are there setups that are more prone to wrong predictions?
- How robust the learning model is when we train and test on the same setups but with a different number of clients, and also when we train and test on a different number of clients with different setups?

In addition to the questions mentioned above, an interesting question may arise concerning how fine-grained the conclusions that we can draw out of learning-based models in case of a server including multiple components are. However, we do not consider this question as we do not have enough data to study this question into details (i.e., the current data show that the Kafka component is always the limiting component for all tested setups). Answering this question can be future work. It would require running new experiments with new workloads to find cases where Spark or Cassandra become the bottleneck component.

4.3.1 Dataset and methodology

The data set is composed of the exported metrics on both hardware level (i.e., resource consumption metrics) and software level (i.e., distributed system software components). For all setups studied in Chapter 3 for each test case (e.g., the Kafka cluster use case), the data set is composed of all setups of that case (i.e., for the Kafka cluster use case, the data set is composed of the 15 setups of the Kafka cluster (see Table 3.10)). Five runs are executed for each setup and we include these runs in the data set (without taking the average of the runs). In order to label the data, we follow the same methodology as explained in Chapter 3 where we identify the bottleneck based on computing $Th_{0.95}$, which is the throughput value within 5% of the observed peak value, and computing $Th_{limit} = Th_{0.95} \times (1 - M)$ where M is an additional safety margin. Based on this methodology, we label the data into three main classes: (i) `client`: this class represents the data points where the client is the bottleneck (the observed throughput is below Th_{limit}), (ii) `gray`: this class represents the data points where tuning (by adding more hardware resources) both the client and the server can improve the overall throughput of the system (the observed throughput is between Th_{limit} and $Th_{0.95}$), and (iii) `server`: this class represents the data points where the server has reached its maximum capacity, i.e., the server is the bottleneck (the observed throughput is above $Th_{0.95}$). We define a baseline which is the accuracy obtained when always predicting the majority class.

For all classifiers, to train models, we use the *Leave-p-out cross-validation schema* [68]. This schema involves using p observations as the test set and the remaining observations as the training set. The experiment is repeated until all subsets p of observations have been considered as part of the test set [82]. In our experiments, the default training schema consists of removing all runs of two setups from the setups set, training the model on all setups but the two removed ones (i.e., the training set), and then performing a prediction on the removed setups (i.e., the test set). This training process is repeated for all setups pairs, and the models average performance on predictions for all setups is reported. More precisely, we report the model accuracy for each test setups set, then we compute the overall average of all accuracies.

Using this default *Leave-p-out* schema allows us to answer the following question: Can we build a model to make predictions on a setup that has never seen before?

In the next steps of the evaluation, we consider other *Leave-p-out* schemas to study other questions, namely:

- Can we build a model that makes predictions on a setup that was part of the training set but with a number of clients it has never seen in any setups?
- Can we build a model that makes predictions for a setup it has never seen and for a number of clients it has never seen in any setups?

.

In this study, we show and compare models accuracy obtained with different subsets of metrics with different classifiers. We also present how important each feature (metric) is to the learning model. Generally, *Feature Importance* (or feature selection) provides a score that indicates how useful or valuable each feature was in the construction of the model. The more an attribute is used to make key decisions, the higher its relative importance. We compute the metrics (feature) importance for all tested cases with the Random Forest classifier. The *Feature Importance* takes a value between $[0, 1]$. The higher this value is, the more important it is.

To build machine learning models, we use Scikit-learn [132], a free machine learning library for the Python programming language. Scikit-learn provides implementation of various classification, regression and clustering algorithms including decision trees, random forests, support vector machines, and logistic regression.

In the following sections, we present the results of our study on the 3 different use cases of the data processing pipeline.

4.3.2 Case study-1: One-tier system (Kafka cluster)

In this section, we apply the learning-based approach to the Kafka cluster use case. We consider the 15 setups of the Kafka cluster described in Table 3.10.

4.3.2.1 What kind of metrics can we use to build an accurate learning model?

In this first experiment, we strive to answer two questions: (i) Can we rely on machine learning techniques to identify the component that is limiting the performance (i.e., client or server) of a Kafka cluster?; (ii) Are there some subsets of metrics that perform better than others in terms of prediction accuracy?

To do so, we build different learning models based on the 6 subsets of metrics mentioned at the beginning of Section 4.3 with 4 classifiers for the Kafka use case using the default *Leave-p-out* schema.

Figure 4.3 shows the average accuracy for different metrics selections with different classifiers for the Kafka cluster use case. The X-axis represents the different metrics selections models with the different classifiers and the Y-axis represents the average model accuracy in predicting whether the Kafka cluster (server) has reached its maximum capacity. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. Based on Figure 4.3, we observe that:

- Generally, most models, except for the *resource consumption* model, achieve better accuracy than the baseline which shows the capacity of high-level metrics to provide information about the bottlenecks in the system.

- Models based on *resource consumption* metrics achieve the worst accuracy for all classifiers, which confirms our findings in Chapter 3 concerning the fact that resource consumption metrics are never reliable indicators of performance bottleneck in the Kafka cluster use case.
- Models based on *recommended* and *server-side-only* metrics achieve the best accuracy for all classifiers compared with the other models. This shows that a combination of reliable metrics is necessary for an accurate prediction of bottlenecks on one hand. On the other hand, it shows that good predictions can be made without the help of client metrics.
- Models based on *all* metrics achieve good accuracy with 2 classifiers (DT and RF) comparing with models that use only specific metrics like the waiting time (WT) and the response time (RT) metrics. This tends to show that a model that combines different types of metrics is more efficient in predicting bottlenecks in a given setup of the Kafka cluster.

Based on these observations, we can rely on machine learning techniques applied to metrics to identify performance bottlenecks in a Kafka cluster as we have seen that different metrics selection models show good prediction accuracy for most classifiers compared with the baseline. Furthermore, models based on *recommended* metrics with the LR classifier and models based on *server-side-only* metrics with the RF classifier show the best accuracy compared with the other models.

Now, we show the accuracy distribution for all models. We only show accuracy distribution with the RF classifier as RF shows the best accuracy for most of the tested models. Figure 4.4 shows the models accuracy distribution of the different approaches with the RF classifier. The X-axis represents the different metrics selections approaches and the Y-axis represents the model accuracy distribution. From Figure 4.4, we observe that:

- Ignoring outliers, the minimum accuracy of all models except the one based on *resource consumption* and the one based on *waiting time* are above the baseline (53%). This confirms that the approach based on classification using hardware and software metrics can learn to identify the bottleneck in the Kafka cluster.
- The distribution of model that uses *recommended* metrics is consistent as the box plot is small.
- Models based on *all*, *recommended*, and *server-side-only* metrics achieve very similar results.

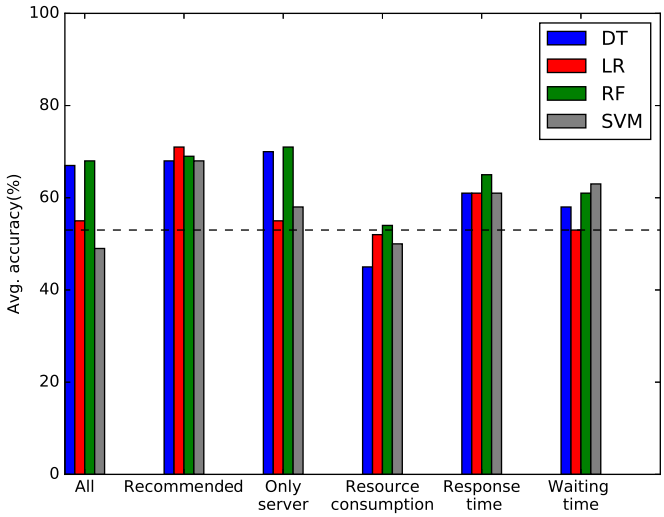


Figure 4.3 – Average accuracy when predicting for a new setup for different metrics selections and different classifiers for the Kafka cluster use case.

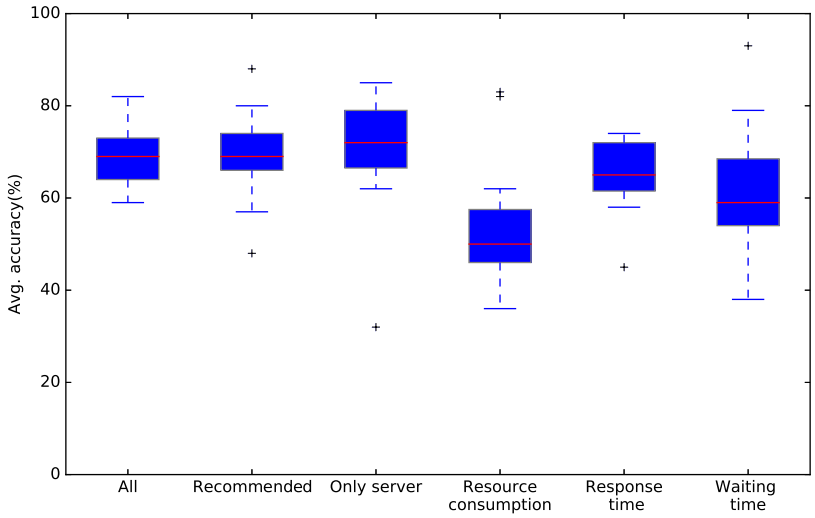


Figure 4.4 – Model accuracy distribution using *Leave-p-out* schema for the Kafka cluster use case with the RF classifier, when predicting on a new setup.

4.3.2.2 What are the important features of the learning model?

In this section, we show the set of features (metrics) that have high importance in building training models based on *all* metrics exported by the Kafka cluster components (client and server) on both hardware and software levels. Figure 4.5 shows the average importance for each feature (metric) that participates in constructing the model that uses *all* metrics with the Random Forest classifier. The X-axis represents how important a feature is and the Y-axis represents the model features (metrics). We export the importance of the features that participate in building the learning model where we use all the metrics as an input to the model. As we have tens of exported metrics, we only show, at most, the top 10 features. From Figure 4.5, we observe that:

- Metrics exported on the client side appear to be in the top 5 of the important features. More precisely, three client metrics appear in the top 5.
- Metrics belonging to the following types (see Table 3.2), appear to be important to the model:
  - **In/Out Data** (i.e., `CL_response_rate`);
  - **Queue Waiting Time** (i.e., `KF_request_queue_time`);
  - **Idle Threads** (i.e., `KF_network_threads_usage`).

  This confirms that metrics of different kinds should be combined to better identify performance bottlenecks for the Kafka cluster.
- The `KF_request_queue_time`, `CL_response_rate`, and `KF_network_threads_usage` metrics are also, reported by the analytical study in Chapter 3 as *reliable* indicators of performance bottleneck for the Kafka cluster. As models based on these kinds of recommended metrics show high prediction accuracy, it implies that a combination of different metrics that act as *reliable* indicators of performance bottleneck is required to build an accurate model.

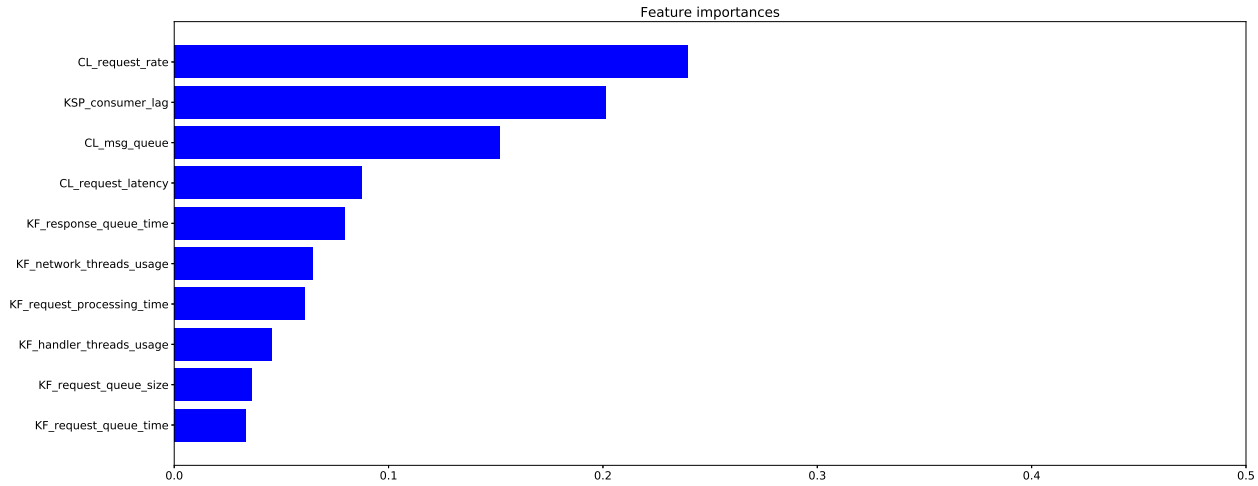


Figure 4.5 – Average feature importance for models based on *all* metrics with the RF classifier for the Kafka cluster.

We also show the important metrics for models based on *server-side-only* metrics. Figure 4.6 shows the importance of server metrics in the learning model with the Random Forest classifier. We show the top 10 features. The X-axis represents how important the feature is in building the learning model and the Y-axis represents the model features (metrics). From Figure 4.6, we observe that:

- Again, metrics of different types (see Table 3.2), are important to the learning model:
  - **Queue Waiting Time** (i.e.,`KF_response_queue_time` and `KF_request_queue_time`);
  - **Queue size** (i.e., `KF_request_queue_size`);
  - **Idle Threads** (i.e., `KF_network_threads_usage`).

- Metrics types mentioned just above are outputted by the analytical study in Chapter 3 as types for some *reliable* indicators of performance bottleneck in the Kafka cluster. Also, we have mentioned above that models based on *server-side-only* metrics show good accuracy in identifying performance bottleneck in the Kafka cluster. This implies that a combination of *reliable* indicators of different types is required to build an accurate learning model to identify performance bottleneck in the Kafka cluster.

Comparing the important features of models based on *all* metrics and models based on *server-side-only* metrics, we notice that ignoring metrics on the client side promotes metrics on the server side that are also *reliable* indicators of performance bottleneck. For instance, the `KF_request_queue_size` metric appears to be in the top-10 of the importance features when the *all* metrics are used in the learning model, while it shows higher importance (i.e., top 3) in models that use *server-side-only* metrics.

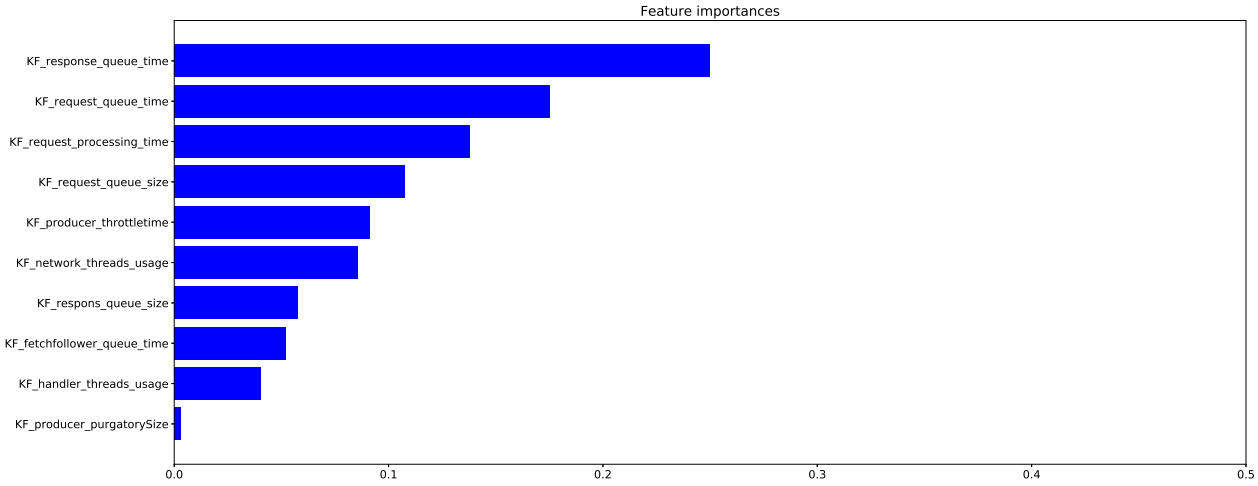


Figure 4.6 – Average importance of metrics for model that uses *server-side-only* metrics features with the RF classifier for the Kafka cluster.

4.3.2.3 Are there setups that are more prone to wrong predictions?

The results presented in Figure 4.3 show that even if the best models manage to correctly identify the bottleneck in the majority of the cases (up to 71% of accuracy), they still do some mistakes in the prediction. In this section, we try to better understand why mistakes are made by trying to identify whether predictions are less accurate for some specific setups. To do so, we apply the following methodology. For the training set, we remove all runs of one setup from the training set. We train the model on all setups but the one removed (i.e., the training set), and then we perform a prediction on the removed setup (i.e., the test set). This testing process is repeated for all setups, and the models performance on predictions for each setup is reported. We apply this methodology on models that use *recommended* metrics as they show the best accuracy compared with other models. Figure 4.7 shows the accuracy for each setup when it is used as a test set. The X-axis represents the different classifiers and the Y-axis represents the model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. For `setup-4` and `setup-14`, the baseline is not visible as the server is always the bottleneck. From Figure 4.7, we notice that:

- The percentage of wrong predictions varies depending on the tested setup and the used classifier.

- For most setups (12/15), the trained models with most classifiers can make predictions with good accuracy compared to the baseline. This leads to the fact that models based on a combination of *reliable* metrics can identify performance bottlenecks in most setups of the Kafka cluster.

- For most classifiers, `setup-4`, `setup-12`, and `setup-14` show a lower accuracy than the baseline. This could be explained by the fact that these three setups have specificities that make it difficult to create a model that is valid for these setups. If we observe the characteristics of the setups (described in Table 3.10), we can notice that `setup-4`, has a specific characteristic that makes its maximum throughput much lower than other configurations. The reason behind that is the use of a single partition, which prevents all parallelism in the processing of messages. Regarding `setup-12` and `setup-14`, it is mostly the fact that they are run on nodes with different hardware configurations (Sagittaire and Ecotype clusters) compared to most experiments which makes it more difficult to achieve high accurate predictions. Along the same line, we observed in the results that for `setup-4`, `setup-12`, and `setup-14`, the server reaches its maximum capacity with a very low number of clients (e.g., in `setup-4`, which is the only configuration with a single partition, the server becomes a bottleneck when 2 clients inject load).

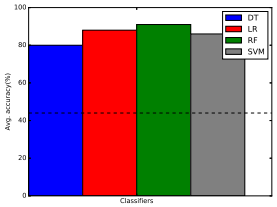
  It is interesting to make the relation between these results and the analysis of reliable metrics presented in Table 3.11. There is no clear conclusion that can be drawn for this comparison. Still, we can notice that some metrics that are classified as reliable indicators for most setups are not reliable for some of these three setups. For instance, the metric `KF_fetchfollower_queue_time` acts as a reliable indicator for most setups but it becomes not reliable for `setup-4`.

  These results tend to show that it is difficult to apply a model that has been trained with experiments that are run on a given platform and obtain good prediction results for other test experiments that are run on a very different platform. More experiments would have to be conducted to confirm this result.

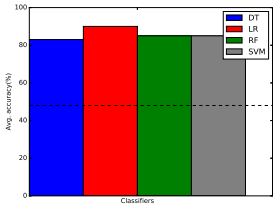
4.3.2.4 Training and testing on the same setup but with different numbers of clients

In this section, we test the capacity of models to generalize to a new number of clients. In this experiment, the tested setup also appears in the training set but with runs involving a different number of clients. This case might happen in practice as the deployment configuration of a Kafka cluster is decided early but the load injected might change while the system is running. We apply the following methodology: For each setup of the Kafka cluster, the training set is built by removing all runs of two number of clients from the examined setup, training the model on all different number of clients but the two removed (i.e., the training set), and then performing a prediction on the removed clients (i.e., the test set). This process is repeated for all numbers of clients pairs, and the models average performance on predictions for all numbers of clients is reported. More precisely, we report the model accuracy for each pair of numbers of clients as a testing set, then we compute the overall average of all accuracies for all setups. We apply this methodology for the different metrics selection approaches with the 4 different classifiers.
Figure 4.8 shows the average accuracy for all setups of the Kafka cluster. The X-axis represents the models with different metrics selections and the Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. From Figure 4.8, we notice that:

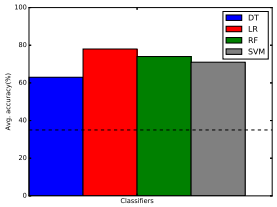
- All models achieve better accuracy than the baseline.



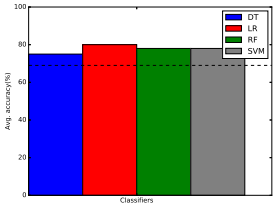
(a) Accuracy for testing setup-0.



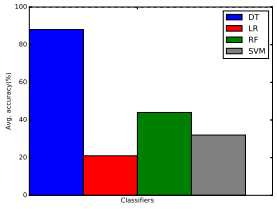
(b) Accuracy for testing setup-1.



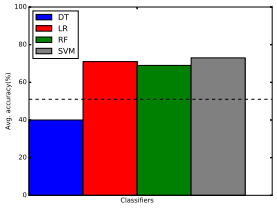
(c) Accuracy for testing setup-2.



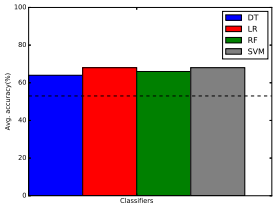
(d) Accuracy for testing setup-3.



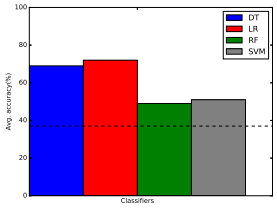
(e) Accuracy for testing setup-4.



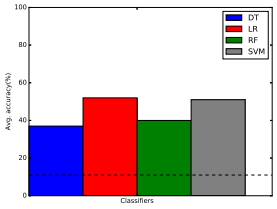
(f) Accuracy for testing setup-5.



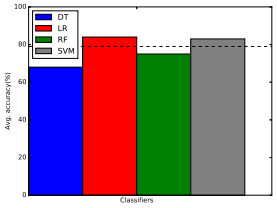
(g) Accuracy for testing setup-6.



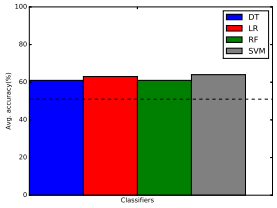
(h) Acuracy for testing setup-7.



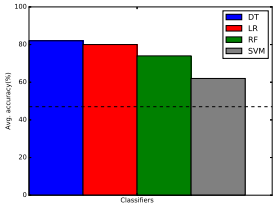
(i) Accuracy for testing setup-8.



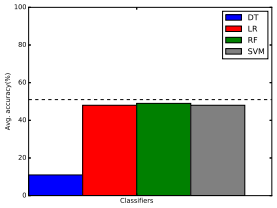
(j) Accuracy for testing setup-9.



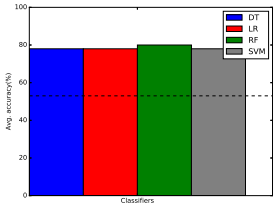
(k) Accuracy for testing setup-10.



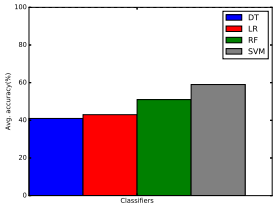
(l) Accuracy for testing setup-11.



(m) Accuracy for testing setup-12.



(n) Accuracy for testing setup-13.



(o) Accuracy for testing setup-14.

Figure 4.7 – Accuracy for all setups of the Kafka cluster when using the recommended metrics with different classifiers.

- Models based on *recommended* metrics are still the most reliable models for all classifiers.
- The best performance is achieved with models based on *server-side-only* metrics and the Random Forest classifier where the accuracy of the predictions reaches 85% on average. This implies that relying on *server-side-only* metrics with a specific classifier is sufficient in a large majority of the cases to identify the bottleneck component in a given setup of the Kafka cluster with a previously unseen number of clients.
- More generally, in cases where the server configuration is already known but with a different number of clients, accuracy above 80% can be achieved.

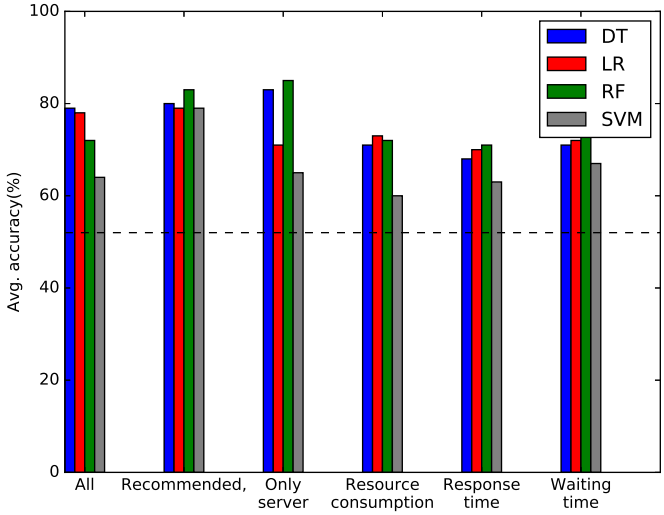


Figure 4.8 – Average accuracy when predicting for a new number of clients for different metrics selections and different classifiers for the Kafka cluster use case.

4.3.2.5 Training and testing on different numbers of clients with different setups

In the previous section, for a known setup, we tested on an unseen number of clients. In this section, we examine a hard case where we test with a number of clients and a setup that never appear in the training set. For example, we have executions for a number of clients equals to 10 for a setup A in the test set and the training set neither contains information about the setup A nor runs with 10 clients for other setups. This case is difficult to learn and to predict as the model has to predict a bottleneck for an unseen number of clients and an unseen setup. The goal here is to check if our model is able to learn and to predict correctly even when it comes to the case with a new number of clients for a new setup.

Figure 4.9 shows the average accuracy of all cases for the different metrics selections models. The *Leave-p-out* methodology with $p = 2$ is still used for running this evaluation. The X-axis represents the different metrics selections models with the 4 different classifiers and the Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. From Figure 4.9, we observe that:

- Only models based on *recommended* and *server-side-only* metrics perform better than the baseline (accuracy up to 66% vs. 53% for the baseline). This implies that the *recommended* and *server-side-only* models are robust when it comes to generalizing to a new number of clients on a new setup.

- Models based on *waiting time* metrics achieve the worst accuracy for most classifiers. This implies that waiting time models are less robust when it comes to generalizing to a new number of clients on a new setup.
- More generally, models based on *RC*, *RT*, and *WT* metrics achieve a lower accuracy than the other models (i.e., *all*, *recommended*, and *server-side-only*). This confirms that combining metrics of different kinds can help in identifying performance bottlenecks in the Kafka cluster.

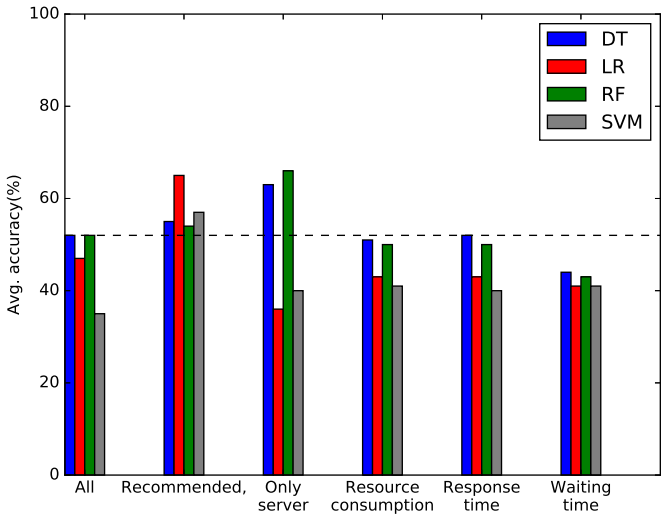


Figure 4.9 – Average accuracy when predicting for a new number of clients and a new setup for different metrics selections and different classifiers for the Kafka cluster use case.

To better understand for this case, where the created models make mistakes in their prediction, we analyze the predictions that are made and compare them to the real class of the tested data. We recall that the class of a data corresponds to the bottleneck in the studied configurations, that is, `client`, `server`, or `gray` zone. For each class, we present the error rate and we further separate the errors in the two other classes. Figures 4.10 (a), (b), and (c) show, for all models with all classifiers, the average error rate for `client`, `gray`, and `server` classes respectively. The X-axis represents the different metrics selections and the Y-axis represents the average error rate. Each bar in the figure represents a classifier. Each bar also shows 3 areas where the bottom of the bar represents the correct prediction and the other 2 parts represents the errors that the model predicts in the other 2 classes. For example, in Figure 4.10 (a), the blue bottom part of each bar represents the average percentage of correct predictions of `clients` being the bottleneck, the gray part represents the average percentage of errors that the model makes by predicting the `gray` class instead of predicting the `client` class, and the red part represents the average percentage of errors that the model makes by predicting the `server`class as bottleneck instead of the`client` class.

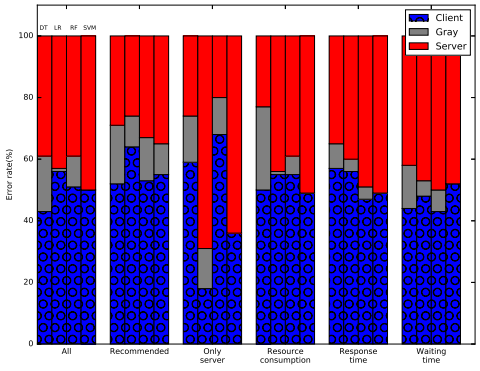
From Figures 4.10 (a), (b), and (c), we see that:

- In general, models based on *server-side-only* metrics with the Random Forest classifier achieve good predictions for `client`, `gray`, and `server` classes.
- Models based on the *recommended* metrics are the best to predict the `client` class.
- In the case of predicting the bottleneck on the server side, models based on *recommended* metrics and *server-side-only* metrics achieve good predictions for most classifiers. Although, models based on

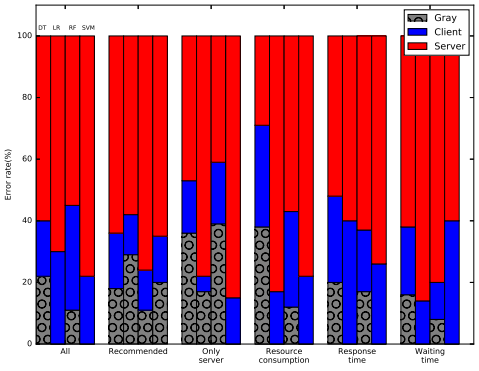
resource consumption metrics achieve good predictions for the `server` class with Random Forest. They mis-predict when it comes to predict the `client` and the `gray` classes.

- Both models based on *response time* and *waiting time* metrics achieve bad accuracy for both the `gray` and the `server` classes for most classifiers.

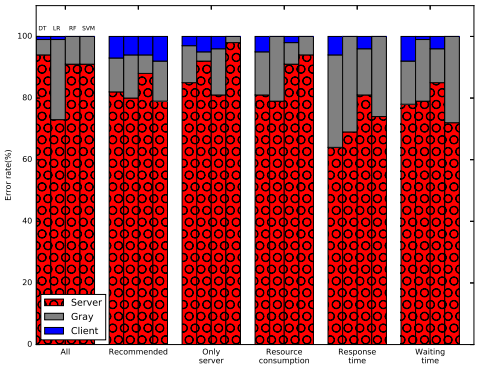
Since the definition of the gray zone is that the throughput can be improved by adding hardware resources on the client and the server sides, we can consider that predicting the `gray` class instead of the `client` or the `server` classes is only a minor error. On Figure 4.10 (c), we can observe that we almost never predict `client` instead of `server`. In Figure 4.10 (a), we observe that some models based on *all*, *recommended*, and *server-side-only* metrics predict `server` instead of `client` in less than 30% of the cases. Finally, Figure 4.10 (b) shows that mistakes are often made when predicting the gray zone. All these results show that the obtained predictions are of better quality than the global results presented in Figure 4.9 may indicate. Another way of looking at the results would be to evaluate that when we would make major mistakes if we decide on a reconfiguration based on the output of the models. A major mistake would be if we decide to allocate more resources to the server while the bottleneck is on the client side, or if we decide not to allocate more resources where the bottleneck is on the server side. Regarding the gray zone, we can assume that any decision taken is correct. Furthermore, we decide for this evaluation, that we allocate more hardware resources to the server when *gray* is predicted. In this case, major errors would only be when predicting *client* instead of *server*, or *server* or *gray* instead of *client*. By taking these assumptions, we see that the best accuracy is achieved with *server-side-only* metrics and the RF classifier and that the percentage of major mispredictions is as low as 10%.



(a) Avg. client error rate.



(b) Avg. gray error rate.



(c) Avg. server error rate.

Figure 4.10 – Average `client`, `gray`, and `server` error rate when predicting for a new number of clients and a new setup for different metrics selections and different classifiers for the Kafka cluster use case.

4.3.3 Case study-2: Two-tier system (Kafka/Spark cluster)

In this section, we apply the learning-based approach to a two-tier system comprising Kafka and Spark Streaming. We run the same experiments as in Section 4.3.2 to answer the questions listed in the introduction of Section 4.3. We consider a Kafka/Spark cluster with 10 setups as described in Table 3.13. In all these setups, we observed that the Kafka servers were the bottleneck of the system when the number of clients was high enough. We consider three classes of predictions: (i) `client`, (ii) `gray`, and (iii) `server`. The `server` class represents the cases where the Kafka component is the bottleneck.

4.3.3.1 What kind of metrics can we use to build an accurate learning model?

Similarly to the Kafka cluster use case, in this experiment, we strive to answer two questions: (i) Can we rely on machine learning techniques to identify the component that is limiting the performance (i.e., client or server) of a Kafka/Spark cluster?; (ii) Are there some subsets of metrics that perform better than others in terms of prediction accuracy? To answer these questions, we apply the same methodology as described for the Kafka use case (Section 4.3.2). Figure 4.11 shows the average accuracy for different metrics selections with different classifiers for the Kafka/Spark cluster use case. The X-axis represents the different metrics selections models with the 4 different classifiers and the Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that

the server is the bottleneck. From Figure 4.11, we observe that:

- For all subsets of metrics, the models obtained with at least two classifiers achieve an accuracy that is higher than the baseline.
- Models based on *recommended* metrics achieve better accuracy than the baseline for all classifiers.
- The best results are obtained with the *recommended* metrics and the LR classifier (accuracy of 72%).
- *Resource consumption* models achieve the worst accuracy for 3 out of 4 classifiers compared with the other models.

Based on these observations, we can rely on machine learning techniques applied to hardware and software metrics to identify performance bottleneck in a Kafka/Spark cluster as we have seen that different metrics selection models show good prediction accuracy with at least 2 classifiers. Furthermore, models based on *recommended* metrics achieve the best accuracy compared with the other models.

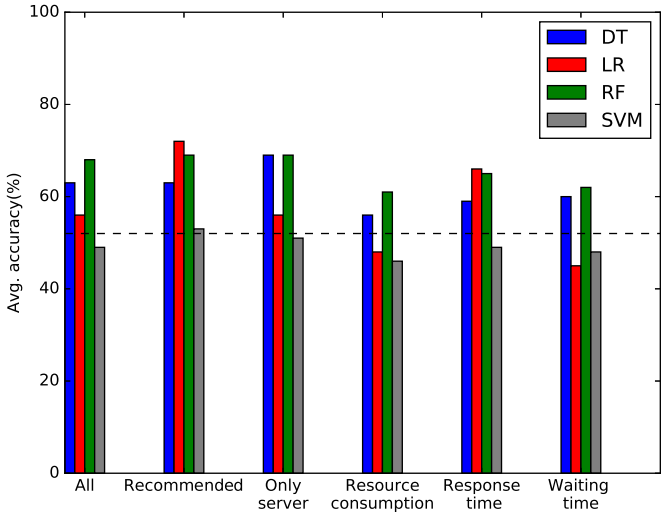


Figure 4.11 – Average accuracy when predicting for a new setup for different metrics selections and different classifiers for the Kafka/Spark cluster use case.

Figure 4.12 shows the models accuracy distribution of the different models with the RF classifier. The X-axis represents the different metrics selection models and the Y-axis represents the model accuracy distribution. We only consider RF as it obtains good results with all subsets of metrics. From Figure 4.12, we see that:

- The minimum accuracy of most models is above the baseline (i.e., 53%).
- The distribution of models based on *recommended*, *sever-side-only*, and *RT* metrics is consistent as the box plots of these models are small.
- The models built based on *recommended* and *server-side-only* metrics achieve very similar results.

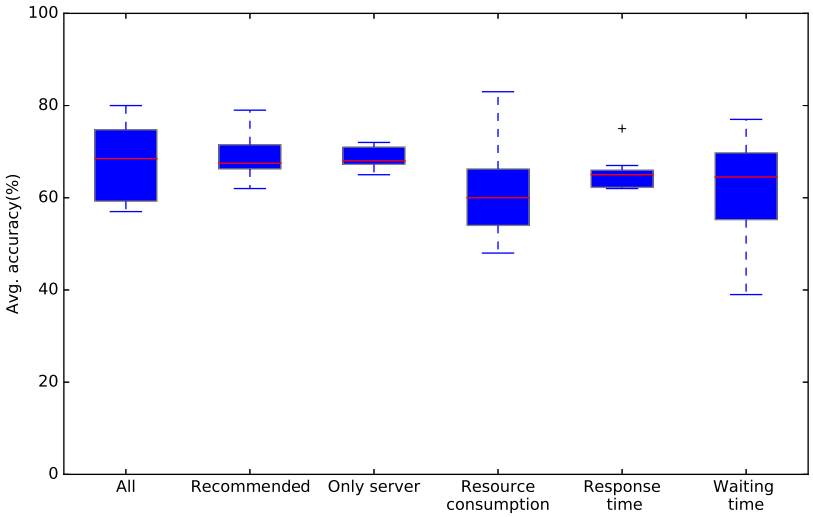


Figure 4.12 – Model accuracy distribution for different metrics selections for the Kafka/Spark cluster use case with the RF classifier, when predicting on a new setup.

4.3.3.2 What are the important features of the learning model?

In this section, we show the importance of metrics in building a learning model. We consider models based on *all* metrics on both hardware and software levels exported by all components of the Kafka/Spark cluster. We consider the RF classifier. Figure 4.13 shows the average importance for each feature (metric) that participates in constructing the learning model. The X-axis represents how important a feature is and the Y-axis represents the model features (metrics). We export the importance of the features that participate in building the learning model where we use all the metrics as an input to the model. As we have tens of exported metrics, we only show, at most, the top 10 features. From Figure 4.13, we notice that:

- Metrics belonging to the following types (see Table 3.2) appear to be at high importance when building the model while the RF classifier is used: **Queue Size** (e.g., `KF_response_queue_size`), **Queue Waiting Time** (e.g., `KF_response_queue_time`), and **Idle Threads** (e.g., `KF_network_threads_usage`).
- Metrics exported by the client and the Kafka servers appear to be more important than metrics exported by Spark. This consistent with the fact that Spark is never the main bottleneck in the system.
- Resource consumption metrics do not appear to be important in building the model.

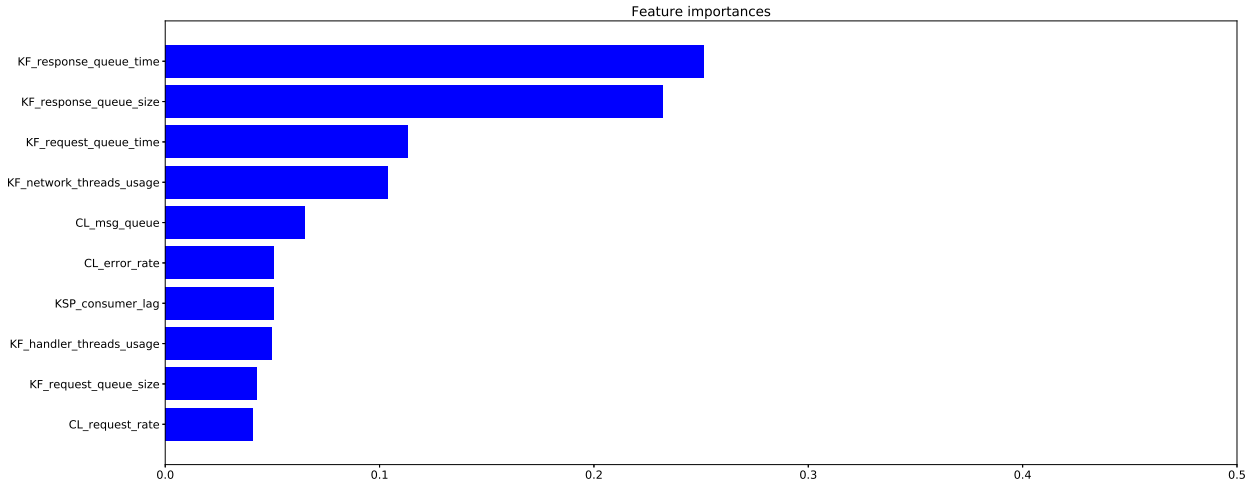


Figure 4.13 – Average importance of the proposed model features with the RF classifier for the Kafka/Spark cluster.

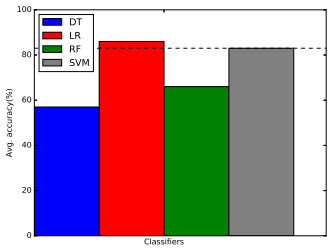
4.3.3.3 Are there setups that are more prone to wrong predictions?

Applying the same methodology as with the Kafka cluster use case, in this section, we try to better understand why mistakes are made by trying to identify whether predictions are less accurate for some specific setups of the Kafka/Spark use case. For the training set, we remove all runs of one setup from the training set. We train the model on all setups but the one removed (i.e., the training set), and then we perform a prediction on the removed setup (i.e., the test set). This process is repeated for all setups, and the models performance on predictions for each setup is reported. We apply this methodology on models that use *recommended* metrics as they achieve the best accuracy compared with other models. Figure 4.14 shows the accuracy for each setup when it is used as a test set. The X-axis represents the different classifiers and the Y-axis represents the model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. From Figure 4.14, we notice that:

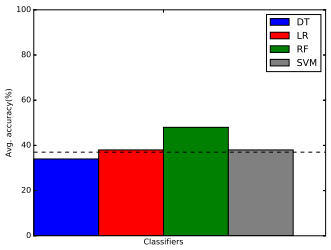
- The percentage of the wrong prediction varies depending on the tested setup and the classifier.
- In general, for all setups, models for most classifiers are able to make predictions with good accuracy compared to the baseline. This leads to the fact that models based on a combination of *reliable* metrics can identify performance bottlenecks in most setups of the Kafka/Spark cluster.
- It is more difficult for the constructed models to achieve high accuracy for setups where the server is almost never the bottleneck (e.g., `setup-1` and `setup-4`).

4.3.3.4 Training and testing on the same setup but with different numbers of clients

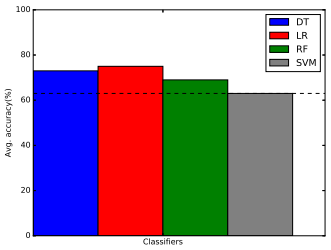
In this section, we test the capacity of models to generalize to a new number of clients. In this experiment, the tested setup also appears in the training set but with runs involving different numbers of clients. We apply the same methodology as described for the Kafka cluster use case (Section 4.3.2.4). For each setup of the Kafka/Spark cluster, the training set is built by removing all runs of two numbers of clients from the



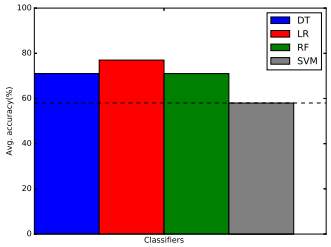
(a) Accuracy for testing setup-0.



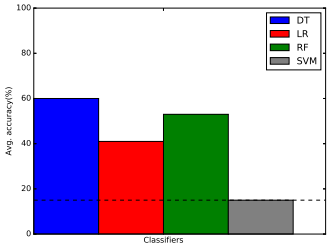
(b) Accuracy for testing setup-1.



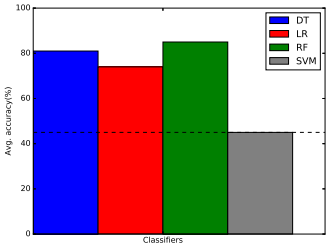
(c) Accuracy for testing setup-2.



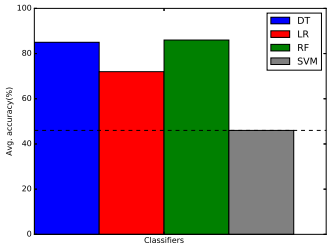
(d) Accuracy for testing setup-3.



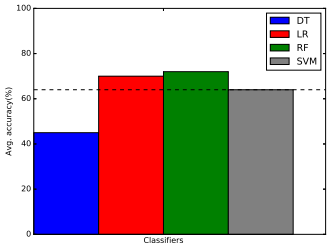
(e) Accuracy for testing setup-4.



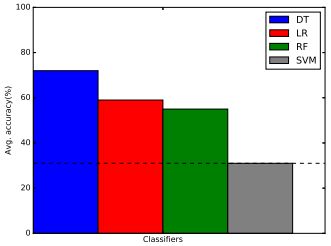
(f) Accuracy for testing setup-5.



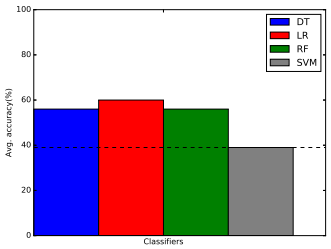
(g) Accuracy for testing setup-6.



(h) Acuracy for testing setup-7.



(i) Accuracy for testing setup-8.



(j) Accuracy for testing setup-9.

Figure 4.14 – Accuracy for all setups of the Kafka/Spark cluster when using the recommended metrics with different classifiers.

examined setup, training the model on all different numbers of clients but the two removed (i.e., the training set), and then performing a prediction on the removed clients (i.e., the test set).

Figure 4.15 shows the average accuracy when predicting for new numbers of clients for each test setup of the Kafka/Spark cluster. The X-axis represents the models with different metrics selections and the Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. From Figure 4.15, we observe that:

- Models based on *all*, *recommended*, and *server-side-only* metrics are very reliable with most tested classifiers.
- The best performance is achieved with models that use *server-side-only* metrics with the RF classifier where the accuracy of the predictions reaches 81% on average.
- Models based on *recommended* metrics achieve (close to) the best accuracy with all classifiers (accuracy up to 80%).
- Models based on *resource consumption*, *response time*, and *waiting time* metrics, obtain an accuracy lower than the baseline. In general, the accuracy achieved with these subsets of metrics is much lower than with the other subsets.

These observations show that models based on *recommended* metrics are robust when it comes to testing on a new number of clients. This implies that these metrics show a significant change in their values and their behaviors with a different number of clients. Moreover, this change corresponds to when the bottleneck shifts from one side to the other. On the other hand, models based on *RC*, *RT*, and *WT* metrics are less robust when it comes to testing on a new number of clients. This means that their metrics do not show significant changes in their values or their behaviors when changing the number of clients or the changes do not correspond to the points when the bottleneck shifts from one side to the other. Thus, relying on these metrics to build prediction models does not guarantee a high prediction accuracy.

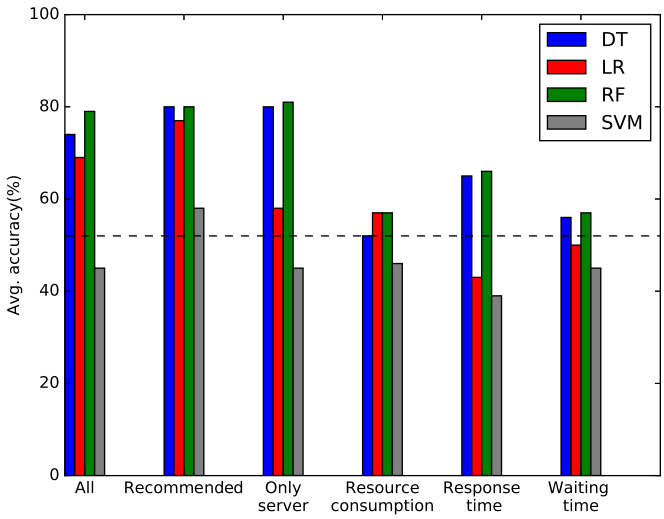


Figure 4.15 – Average accuracy when predicting for a new number of clients for different metrics selections and different classifiers for the Kafka/Spark cluster use case.

4.3.3.5 Training and testing on different numbers of clients with different setups

In this section, we examine the robustness of the learning models when it comes to predicting a difficult case where both the number of clients and the setup have not been seen before by the model. We apply the same methodology described for the Kafka cluster (Section 4.3.2.5). Figure 4.16 shows the average accuracy for testing on new numbers of clients with new setups for different metrics selections models. The X-axis represents the different metrics selections models with the 4 different classifiers and the Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. From Figure 4.16, we observe that:

- Models based on *all*, *recommended*, and *server-side-only* metrics perform better than the baseline (i.e. 53%) for most classifiers (i.e., 3/4 classifiers). However, Models based on *resource consumption*, *response-time*, and *waiting-time* metrics perform worse than the baseline for most classifiers. This implies that relying on one kind of metric is not sufficient to identify performance bottlenecks in the Kafka/Spark cluster. Also, these models show less robustness when it comes to testing on a case where both the number of clients and the setup are unseen before.
- Models based on *recommended* metrics achieve the best accuracy for most classifiers comparing with the other models.
- Model based on *resource consumption* metrics achieve the worst accuracy for most classifiers comparing with the other models and the baseline. This confirms that resource consumption metrics are not good metrics to identify the bottleneck component in the Kafka/Spark cluster.

Based on these observations, models based on *recommend* and *server-side-only* are more robust to predict bottlenecks of a new case where both the number of clients and the setup configuration have not seen before by the model.

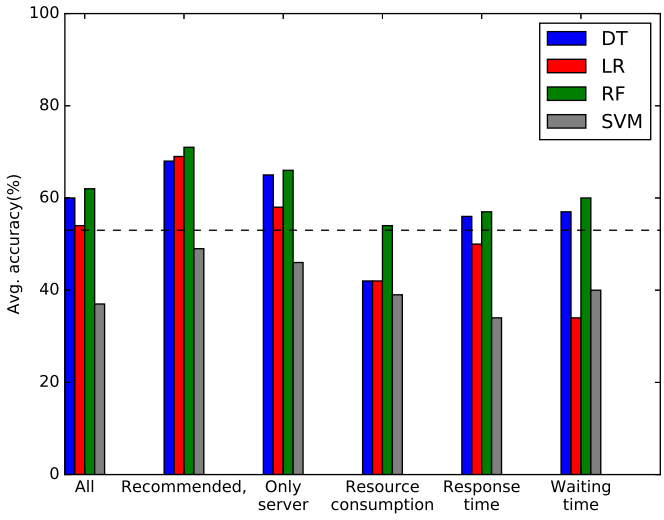


Figure 4.16 – Average accuracy when predicting for a new number of clients and a new setup for different metrics selections and different classifiers for the Kafka/Spark cluster use case.

To better understand for this case where the created models make mistakes in their prediction, we analyze the predictions that are made and compare them to the real class of the tested data. We recall that the class

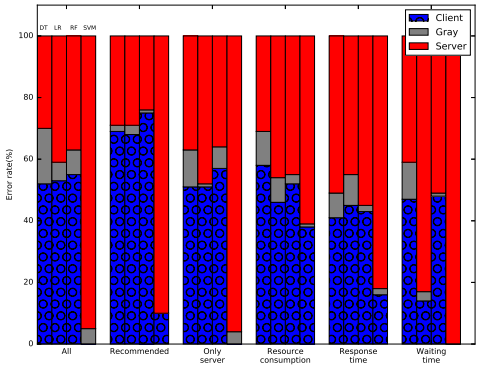
of a data corresponds to the bottleneck in the studied configurations, that is, `client`, `server`, or `gray` zone. For each class, we present the error rate and we further separate the errors in the two other classes.

Figures 4.17 (a), (b), and (c) show, for all models with all classifiers, the average error rate for `client`, `gray`, and `server` classes respectively. The X-axis represents the different metrics selections and the Y-axis represents the average error rate. Each bar in the figure represents a classifier. Each bar also shows 3 areas where the bottom of the bar represents the correct prediction and the other 2 parts represents the errors that the model predicts in the other 2 classes.

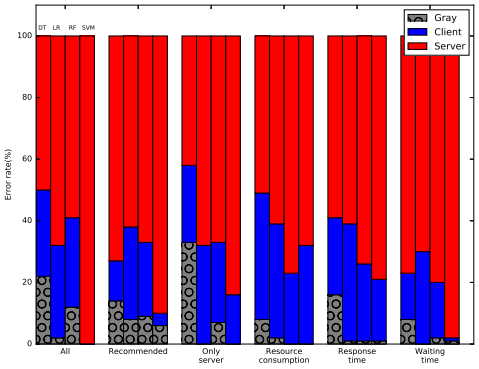
From Figures 4.17 (a), (b), and (c), we observe that:

- In general, models based on *recommended* metrics achieve good prediction for `client` and `server` classes for most classifiers.
- In the case of predicting the bottleneck on the server side, models based on *all*, *recommended*, and *server-side-only* metrics achieve better prediction accuracy than *RC*, *RT*, and *WT* models for all classifiers.
- In the case of predicting the bottleneck on the client side, models based on *recommended* metrics achieve the best prediction accuracy for most classifiers.

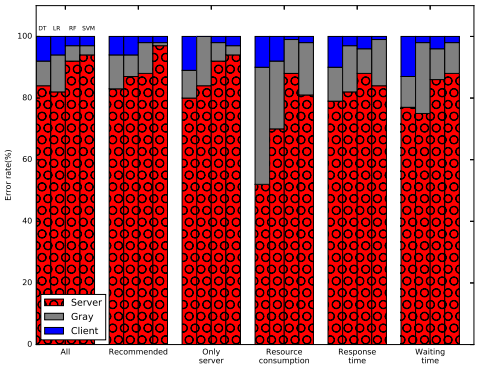
On Figure 4.17 (c), we can observe that we almost never predict the `client` class instead of the `server` class. In Figure 4.17 (a), we observe that some models based on *recommended* metrics predict the `server` class instead of the `client` class in less than 30% of the cases. Finally, Figure 4.17 (b) shows that mistakes are often made when predicting the gray zone. Taking the assumptions described in the Kafka use case (Section 4.3.2.5), we see that the best accuracy is achieved with *recommended* models and the RF classifier, and that the percentage of major mispredictions is as low as 5%.



(a) Avg. client error rate.



(b) Avg. gray error rate.



(c) Avg. server error rate.

Figure 4.17 – Average `client`, `gray`, and `server` error rate when predicting for a new number of clients and a new setup for different metrics selections and different classifiers for the Kafka/Spark use case.

4.3.4 Case study-3: Multi-tier system (Kafka/Spark/Cassandra cluster)

In this section, we apply the learning-based approach to a multi-tier system comprising Kafka, Spark Streaming, and Cassandra. We run the same experiments as in Sections 4.3.2 and 4.3.3 to answer the questions listed in the introduction of Section 4.3. We consider the 10 setups of the data processing pipeline described in Table 3.16. In all these setups, we observed that the Kafka servers were the bottleneck of the systems when the number of clients was high enough. We consider three classes of predictions: (i) `client`, (ii) `gray`, and (iii) `server`. The `server` class corresponds to the case where Kafka is the bottleneck.

4.3.4.1 What kind of metrics can we use to build an accurate learning model?

As for the previous cases, we start by training models to make predictions on a new setup. The same methodology as before is applied. Figure 4.18 shows the average accuracy for the different models with different classifiers for the data processing pipeline. The X-axis represents the different metrics selections models with the 4 different classifiers and the Y-axis represents the average model accuracy. As with the full pipeline, the number of cases where the server is the bottleneck is low, we choose as a baseline the accuracy obtained when always predicting that the client is the bottleneck (unlike the two previous cases, i.e., the Kafka case and the Kafka/Spark case).

Based on Figure 4.18, we see that:

- All models achieve better accuracy than the baseline (i.e., 57%) for all classifiers.
- The best results are obtained with the *server-side-only* metrics and the RF classifier (accuracy of 74%).
- *Resource consumption* models achieve a low accuracy for all classifiers.

Based on these observations, we can rely on machine learning techniques applied to hardware and software metrics to identify performance bottlenecks in the full data processing pipeline as we see that different metrics selection models show good prediction accuracy for all classifiers compared with the baseline. However, models based on *server-side-only* metrics with the RF classifier achieve the best accuracy compared with the other models.

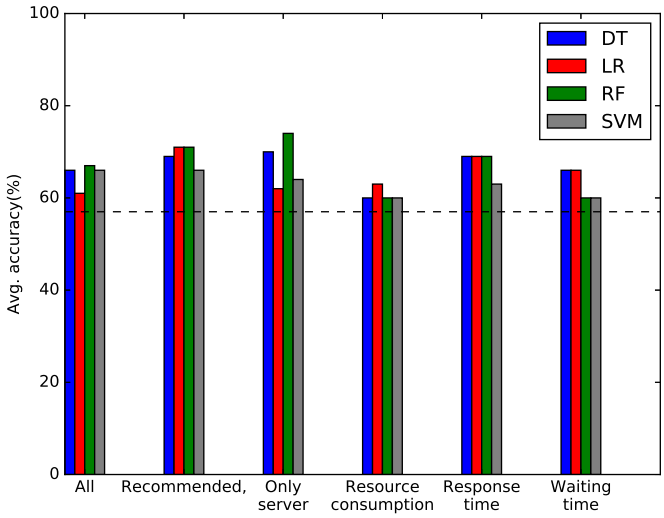


Figure 4.18 – Average accuracy when predicting for a new setup for different metrics selections and different classifiers for the full data pipeline use case.

We show the accuracy distribution for the different models. As in the previous cases, RF shows the best accuracy for most of the tested models, we only show the models accuracy distribution with the RF classifier. Figure 4.19 shows the models accuracy distribution of the different metrics selections with the RF classifier. The X-axis represents the different metrics selections models and the Y-axis represents the model accuracy distribution. The results presented in Figure 4.19 tend to show that predictions are more difficult to make in this case as all models have a minimum accuracy below the baseline (i.e., 57%). Also, the quality of the predictions tends to vary a lot as the box plots are large.

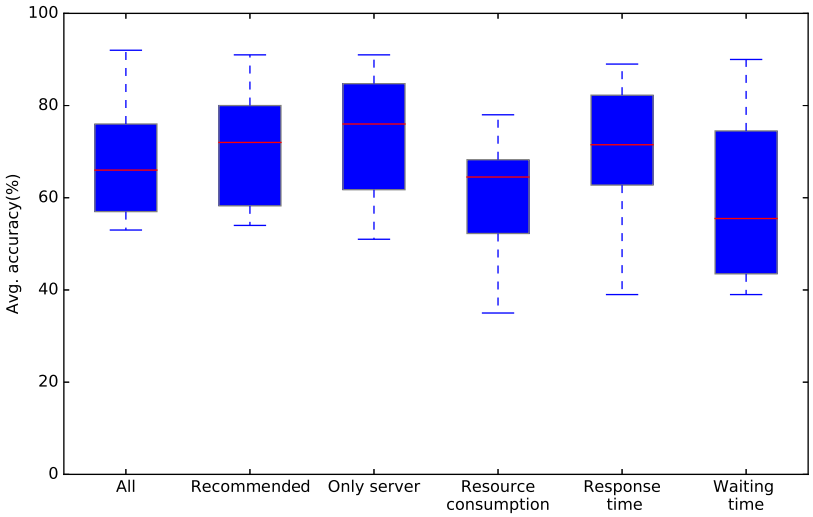


Figure 4.19 – Model accuracy distribution for different metrics selections for the full data processing pipeline use case with the RF classifier, when predicting on a new setup.

4.3.4.2 What are the important features of the learning model?

In this section, we show the importance of metrics in building a learning model. We consider models based on *all* metrics on both hardware and software levels exported by all components of the data processing pipeline. We consider the RF classifier. Figure 4.20 shows the average importance for each feature (metric) that participates in constructing models. The X-axis represents how important a feature is and the Y-axis represents the model features (metrics). We export the importance of the features that participate in building the learning model where we use all the metrics as an input to the model. We only show the top 10 features. Based on Figure 4.20, we observe that:

- Metrics belonging to the following types have an impact in building the training model when the RF classifier is used: **Queue Waiting Time** (e.g., `KF_request_queue_time`), and **Idle Threads** (e.g., `KF_network_threads_usage`), and **Error Rate** (e.g., `CL_error_rate`).
- The importance of some metrics changes depending on the studied case (i.e., Kafka, Kafka/Spark, and Kafka/Spark/Cassandra). Some metrics are important in all studied cases (e.g., `KF_response_queue_time`), while others become less important (e.g., `KF_network_threads_usage`). For instance, `CL_request_rate` appears to be in the top 5 for the Kafka/Spark/Cassandra cluster, while it is a very important metric in the case of the Kafka cluster, however it has a minor importance in the Kafka/Spark use case. Also, `CL_msg_queue` is a very important metric in the case of the Kafka cluster and the Kafka/Spark cluster, but it has a minor importance in the Kafka/Spark/Cassandra use case.

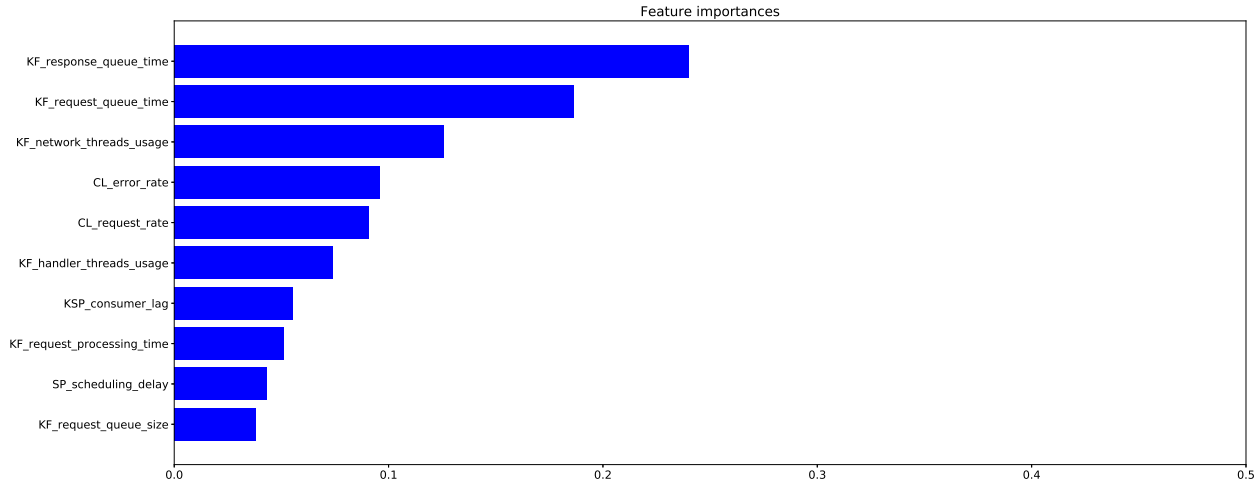


Figure 4.20 – Average importance of the proposed model features with the RF classifier for the full data processing pipeline use case.

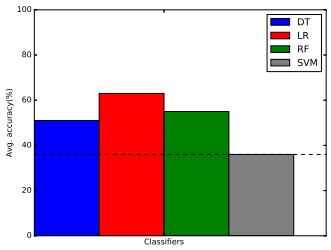
4.3.4.3 Are there setups that are more prone to wrong predictions?

In this section, we try to answer the question concerning if there are some setups always lead to a wrong prediction when they are in the testing sets. To do so, we apply the same methodology as described in the Kafka and Kafka/Spark use cases (Sections 4.3.2.3 and 4.3.3.3 respectively). We consider models based on *recommended* metrics with all classifiers.

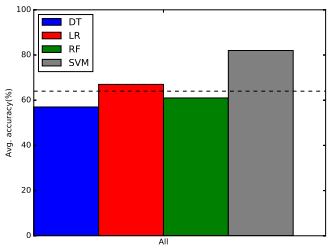
Figure 4.21 shows accuracy for each setup when it is used as a test set. The X-axis represents the different classifiers and the Y-axis represents the model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the client is the bottleneck.

From Figure 4.21, we notice that:

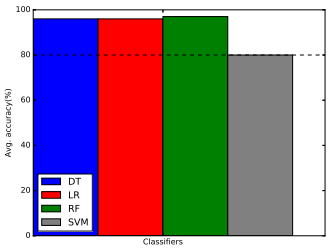
- Some setups are more prone to wrong predictions when the baseline represents the accuracy obtained when always predicting that the client is the bottleneck. This is especially the case for `setup-1`, `setup-4`, and `setup-5`.
- For all setups, with most classifiers, models achieve an accuracy similar to or better than the baseline (i.e., 57%). This implies that models based on *recommended* metrics are robust when it comes to predicting new setups.
- The highest accuracy is achieved with `setup-2` (above 85%) for all classifiers while the lowest accuracy is achieved with `setup-7` (below 40%). In fact, `Setup-2` and `setup-7` are very different. They differ from the point of view of the number of Spark executors (12 vs. 3), the Cassandra consistency level (all vs. 1) and the workload (WC vs. FDP). As a consequence, in `setup-7`, the server is often the bottleneck whereas in `setup-2` it is mostly the client. Since there are a few cases where the server is the bottleneck, it will be difficult for models to predict this class (i.e., the server class). Furthermore, we can note two following points: (i) `KF_response_queue_time` and `KF_request_queue_time` are most important metrics for this use case according to Figure 4.20, (ii) these metrics are both *misleading* for `setup-7` according to Table 3.17.



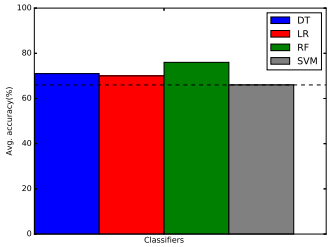
(a) Accuracy for testing setup-0.



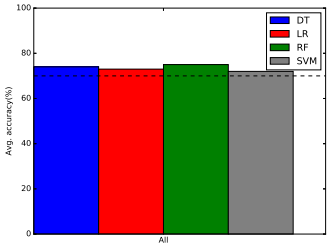
(b) Accuracy for testing setup-1.



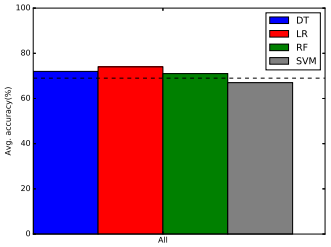
(c) Accuracy for testing setup-2.



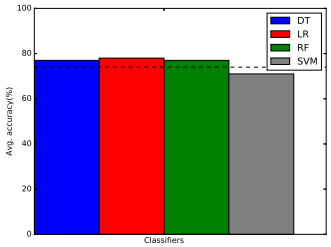
(d) Accuracy for testing setup-3.



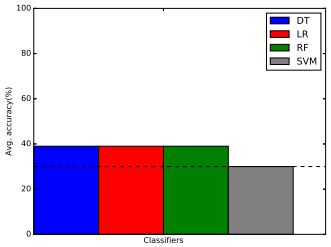
(e) Accuracy for testing setup-4.



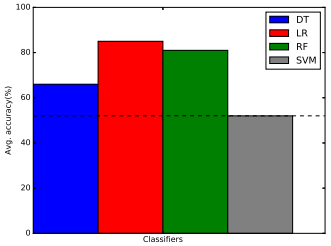
(f) Accuracy for testing setup-5.



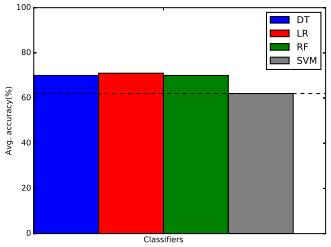
(g) Accuracy for testing setup-6.



(h) Acuracy for testing setup-7.



(i) Accuracy for testing setup-8.



(j) Accuracy for testing setup-9.

Figure 4.21 – Accuracy for all setups of the Kafka/Spark/Cassandra cluster when using the recommended metrics with different classifiers.

4.3.4.4 Training and testing on the same setup but with different numbers of clients

In this section, we test the capacity of models to generalize to a new number of clients. We apply the same methodology described for the Kafka cluster use case (Section 4.3.2.4).

Figure 4.22 shows the average accuracy for each test setup of the Kafka/Spark/Cassandra cluster. The X-axis represents the models with different metrics selections and the Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the client is the bottleneck. The conclusions from Figure 4.22 are similar to the ones drawn for the experiments with the Kafka cluster (Figure 4.8) and with the Kafka/Spark cluster (Figure 4.15). The main observation is that models that combine different types of metrics (*all*, *recommended*, and *server-side-only*) perform better than models that focus on a single kind of metrics (*RC*, *RT*, and *WT*).

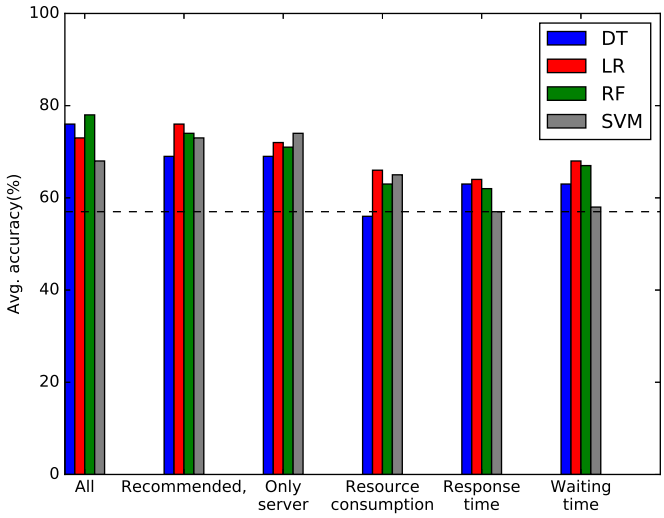


Figure 4.22 – Average accuracy when predicting for a new number of clients for different metrics selections and different classifiers for the full data processing pipeline use case.

4.3.4.5 Training and testing on different numbers of clients with different setups

Similarly to the Kafka cluster use case and the Kafka/Spark use case, in this section, we check the robustness of the learning models by testing a difficult case where both the number of clients and the setup have not been seen before by the model. We apply the same methodology as described for the Kafka cluster (Section 4.3.2.5).

Figure 4.23 shows the average accuracy for testing on new numbers of clients with new setups for different metrics selections models. The X-axis represents the different metrics selections models with the 4 different classifiers and the Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the client is the bottleneck. From Figure 4.23, we observe that:

- Models based on *recommended* metrics achieve the best accuracy (accuracy up to 71%).
- All models perform better than the baseline (i.e., 57%) for at least two classifiers.

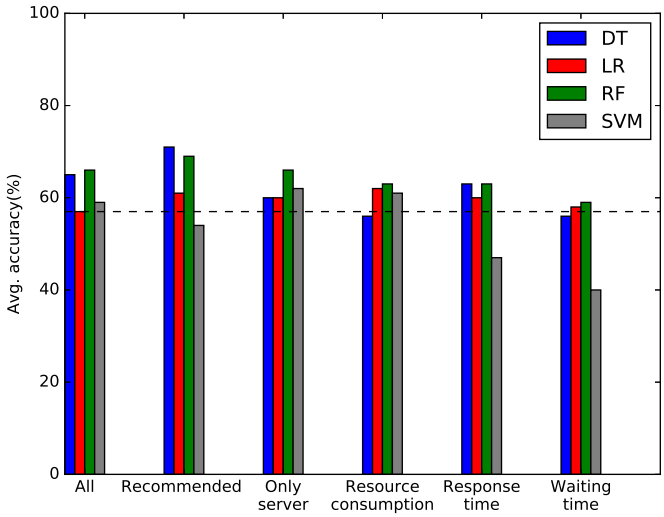


Figure 4.23 – Average accuracy when predicting for a new number of clients and a new setup for different metrics selections and different classifiers for the full data processing pipeline use case.

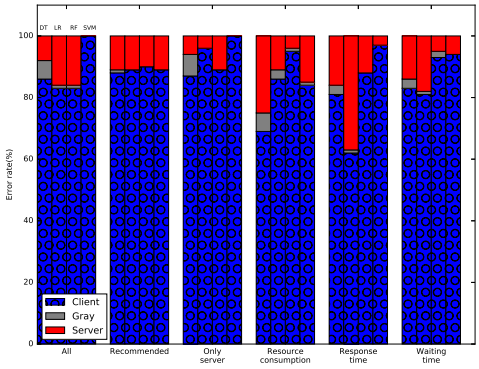
To better understand for this case, where the created models make mistakes in their predictions, we analyze the predictions that are made and compare them to the real class of the tested data. We recall that the class of a data corresponds to the bottleneck in the studied configurations, that is, `client`, `server`, or `gray` zone. For each class, we present the error rate and we further separate the errors in the two other classes.

Figures 4.24 (a), (b), and (c) show, for all models with all classifiers, the average error rate for `client`, `gray`, and `server` classes respectively. The X-axis represents the different metrics selections and the Y-axis represents the average error rate. Each bar in the figure represents a classifier. Each bar also shows 3 areas where the bottom of the bar represents the correct prediction and the other 2 parts represents the errors that the model predicts in the other 2 classes.

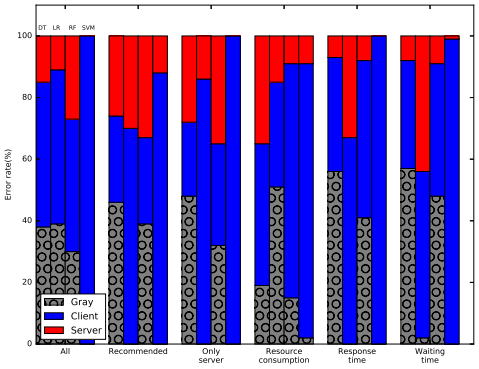
From Figures 4.24 (a), (b), and (c), we observe that:

- In general, all models achieve good prediction for the `client` class for most classifiers. Also, all models achieve good prediction for the `server` class if we would interpret in practice the `gray` class as the server is the bottleneck.
- Mostly, models achieve a much lower accuracy when it comes to predicting the `server` class compared to the previous use cases. It can be explained by the fact that the dataset includes much fewer examples where the server is the bottleneck compared to the previous use cases.

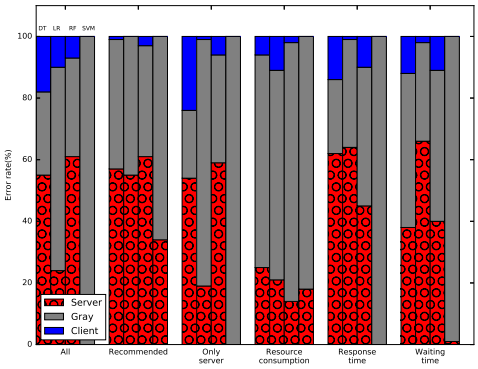
Applying the same reasoning as in the previous section (Section 4.3.2.5) about the major mistakes, the results become again much better. For all models, the number of major mispredictions is very small. Best results are achieved with *server-side-only* metrics and the SVM classifier where no mistakes are made.



(a) Avg. client error rate.



(b) Avg. gray error rate.



(c) Avg. server error rate.

Figure 4.24 – Average `client`, `gray`, and `server` error rate when predicting for a new number of clients and a new setup for different metrics selections and different classifiers for the full data processing pipeline use case.

4.3.5 Summary of the comparison of metrics selections

In addition to building an accurate prediction model, the main goal of this section was to compare the accuracy of the predictions that can be achieved when considering different types of metrics as input for the models. We summarize the results obtained from this point of view in two tables. Table 4.1 shows the subset of metrics that give the best accuracy for each prediction scenario for all use cases of the data processing pipeline. Table 4.2 summarizes the metrics types that classifiers select as their main features for making predictions. To build this table we consider the top 5 features.

The results from our study show that:

- Models based on *recommended* metrics achieve the best accuracy for a majority of classifiers for the three uses cases of the data processing pipeline. This shows that identifying and combining *reliable* metrics is important to provide accurate information about the bottlenecks in the system (Table 4.1).
- Models based on *resource consumption*, *response time*, and *waiting time* metrics achieve a lower accuracy compared with the baseline and the other models for all studied cases of the data processing pipeline. This confirms that combining metrics of different kinds can help in identifying performance bottlenecks in the data processing pipeline.

- Table 4.2 confirms some of the findings from related works. The *Idle Threads*, *Queue Waiting Time*, and *In/Out Data* types are useful to analyze the behavior of a multi-tier system and to identify its bottlenecks. However, metrics from these different kinds should be combined to get accurate results. On the other hand, Table 4.2 confirms that resource consumption metrics are of little help in our studied use cases.

	Kafka cluster	**Kafka/Spark cluster**	**Kafka/Spark/Cassandra cluster**
Test on a new setup			
Model	Recommended, server-side-only	Recommended	Server-side-only
Test on a new number of clients			
Model	Server-side-only	Recommended, server-side-only	All
Test on a new number of clients and a new setup			
Model	Recommended, server-side-only	Recommended	Recommended

Table 4.1 – Summary of models that give the best accuracy for each use case of the data processing pipeline.

Metric Type	**Kafka cluster**	**Kafka/Spark cluster**	**Kafka/Spark/Cassandra cluster**
Resource Consumption (RC)			
Idle Threads (ITD)		✓	✓
Error Rate (ER)			✓
Queue Waiting Time (QWT)	✓	✓	✓
Queue Size (QS)		✓	
Latency (LY)	✓		
Processing Time (PT)			
In/Out Data (IOD)	✓	✓	✓
Uncategorized (UC)			

Table 4.2 – Summary of top 5 metrics types for all use cases of the data processing pipeline.

4.4 Conclusion

In this chapter, we presented a learning-based approach for performance bottleneck identification in a data processing pipeline. We considered three use cases of the data processing pipeline including: a one-tier system (Kafka cluster), a two-tier system (Kafka/Spark Streaming), and a multi-tier system (full data processing pipeline).

Comparing learning models based on different selections of metrics, we evaluated the capacity of prediction models to generalize to new setups, to new numbers of clients, and even to cases where both the setup and the number of clients are new. Our analysis was run considering 4 different machine learning classifiers (Decision Tree, Random Forest, Support Vector Machines, and Logistic Regression) to avoid drawing conclusions that would be specific to one classifier. The main results we obtained are consistent over the three considered use cases. The main results can be summarized as follows:

- We manage to achieve good predictions, up to 74%, with models based on *recommended* and *server-side-only* metrics when predicting on a new setup.
- Best results are obtained when generalizing to a new number of clients (accuracy up to 85%). Even in the difficult case where we try to predict on a new number of clients and a new configuration, we

manage to reach at best an accuracy of 71% . The best prediction, in this case, is achieved by models based on *recommended* and *server-side-only* metrics.

- When analyzing the results in more detail, we see that if we decide to consider the server side as the bottleneck as soon as we predict that we are entering a gray zone where it is unclear whether the server has already reached its maximum capacity, we almost always make the right decision for cases where the server side is the bottleneck.

- The comparison of the results obtained by considering different subsets of metrics as input for the prediction models show that a combination of metrics of different types guarantees a good prediction of performance bottleneck in the data processing pipeline use case.

- Metrics of types: *Idle Threads (ITD)*, *Queue Waiting Time (QWT)*, and *In/Out Data (IOD)* are important but should be combined. On the other hand, models based on *resource consumption* metrics almost never achieve good accuracy in predicting the bottleneck in our studied use cases.

The results presented above are consistent with our main conclusions from Chapter 3. Furthermore, the works in these two chapters are complementary as the models based on metrics recommended by the results of Chapter 3 are the ones that achieve most often the best accuracy in the predictions. The good results obtained by *server-side-only* models also confirm the results of Chapter 3 that there are enough reliable indicators on the server side to correctly identify performance bottleneck in the data processing pipeline. By studying into detail the setups where more mistakes are made in the predictions, we noticed that having significant changes in the underlying hardware characteristics was making it difficult to predict the bottleneck component accurately. Further experiments and analyses would need to be conducted to better understand this result.

The main question that remains at the end of this depth study is to understand if the results we obtained on the data processing pipeline are specific to this use case or if they are more generally valid for multi-tier applications. To get a better understanding of this point, we apply in the next chapter our analytical study and our learning-based approach to another multi-tier system. Namely, we consider a LAMP stack comprising an Apache server that executes PHP code and interacts with a MySQL database.

Chapter 5

Towards a General Approach for Bottleneck Detection in Multi-Tier Distributed Systems: Studying a Web Stack

Contents

In this chapter, we consider a Web stack, LAMP, where we apply the analytical study presented in Chapter 3 and the learning-based approach presented in Chapter 4. The goal of this chapter is to (i) check the robustness of the two proposed approaches when examining a different system in terms of characteristics and workload, and (ii) investigate if it is possible to generalize the findings obtained on the data processing pipeline to other systems. The main results obtained on the data processing pipeline, that we would like to validate on the LAMP stack are:

- Resource consumption metrics are not reliable indicators of performance bottlenecks in multi-tier systems.
- A small set of *reliable* metrics is usually enough to identify if the server reached its maximum capacity in a large number of setups.
- A model can be built to reliably identify bottlenecks in a multi-tier system. To achieve high accuracy, such a model should use as input a combination of metrics of different types.

We start this chapter by presenting the background related to the LAMP stack and its architectural design in Section 5.1. Section 5.2 presents the results of the analytical study for this use case. After that, we apply our approach based on machine learning to build an automatic tool to identify the bottleneck in the studied use case in Section 5.3. Finally, we discuss the findings of the two approaches and we draw some conclusions in Section 5.4.

5.1 Background

In this section, we describe the architectural design of the LAMP stack including its tiers/components. For each tier/component, we present the main design principles, provided features, and factors that affect their performance. We also present the role of each component and how it contributes to the studied case. We give examples from production systems of where and how these software components are used. Finally, we present the main performance metrics exported by each software component in the stack.

5.1.1 LAMP stack: Architectural design

The LAMP stack is a web stack based on the following components: the Linux operating system; An Apache HTTP server [98], a widely used web server software; MySQL [135], a relational database management system; PHP [140], a scripting programming language (see Figure 5.1). In the following sections, we present in detail the two main components of the LAMP stack: Apache server and MySQL.

5.1.1.1 Apache server

Apache HTTP server [98] is widely used as a Web server software. It is a secure, efficient and extensible server that provides HTTP services corresponding to the current HTTP standards. At its most basic level, an Apache server processes clients requests and delivers responses back to clients. The traditional Apache structure is based on a single parent process and a group of reusable children (see Figure 5.2). The parent reads the configuration and manages the pool of children. Each child at any time is either serving a single request or sleeping.

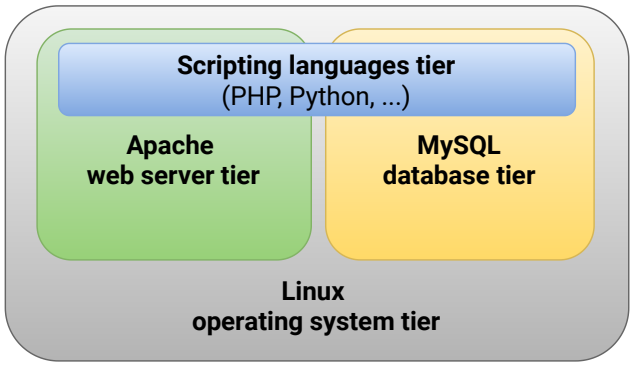


Figure 5.1 – The LAMP stack [55].

Apache server has a modular architecture where the core provides the basic functionality and separate sets of modules are supported for handling HTTP requests. Also, it follows an *implicit invocation* [1] architectural style [63]. Apache supports multiple modes of operation (multi-process, multi-thread, etc.) and provides a large set of features [61]. Apache relies on `Multi-Processing Modules (MPMs)` to determine how to use resources (threads/processes, ports, etc.) to process clients requests. The three main MPMs for Unix-like systems are `prefork, worker, and event` [43].

- `Prefork MPM`: In the prefork MPM, the parent process uses `fork()` to prefork a pool of child processes at startup time. Each idle child process joins a queue to listen for incoming requests. This MPM uses a method called accept mutex (Mutual Exclusion mechanism) to ensure that only one process listens for and accepts the next TCP request. The first idle worker process in the queue acquires the mutex and listens for the next incoming connection. After receiving a connection, it releases the accept mutex by passing it to the next idle process in the queue and processes the request (during which time it is considered a busy worker). After it finishes processing the request, it joins the queue once again. Because this MPM needs a higher number of processes to handle any given number of requests, it is generally more memory-hungry than multi-threaded MPMs like `worker` and `event`.

- `Worker MPM`: In the worker MPM, the parent process creates a certain number of child processes, and each child process creates a constant number of threads (`ThreadsPerChild`), as well as a listener thread. Like the prefork MPM, the worker MPM uses an accept mutex to designate which thread will process the next incoming request. Each child process listener thread only joins the idle queue (which indicates that it is eligible to obtain the accept mutex) if it detects that at least one worker thread within the child process is currently idle. Therefore, for every process that has at least one idle worker thread, the listener thread will join the queue to apply for an accept mutex. Unlike the prefork MPM, the worker MPM enables each child process to serve more than one request at a time, by utilizing multiple threads. Because you only need one thread per connection instead of forking one process per connection, this MPM tends to be more memory-efficient than the prefork MPM.

- `Event MPM`: In the event MPM, each child process creates multiple threads (determined by `ThreadsPerChild`), in addition to one listener thread. In the other types of MPMs, if a worker thread or process handles a request on a keep-alive connection, the connection stays open (and the worker is blocked from processing other requests) for the duration of the `KeepAliveTimeout`, or until the client closes the connection. A major benefit of the event MPM is that it handles keep-alive connections

[1] Implicit invocation is a style of software architecture in which a system is structured around event handling.

more efficiently by utilizing the kernel I/O methods like `epoll` (on Linux). Moreover, a worker thread can send a response to the client, and then pass control of the socket to the listener thread, freeing the worker to serve another request. Once an event is detected, the listener thread passes it on to the next available idle worker thread. If the `KeepAliveTimeout` is reached before any activity occurs on the socket, the listener thread closes the connection. Also, if any listener thread detects that all worker threads within its process are busy, it will close keep-alive connections, forcing clients to create new connections that can be processed more quickly by other processes' worker threads. Because the dedicated listener thread helps monitor the lifetime of each keep-alive connection, worker threads that would otherwise have been blocked (waiting for further activity) are instead free to server other active requests.

In our study, we use the `Event MPM` as it is the default mode for most modern platforms. Apache performance can be impacted by many factors including the maximum number of connections that are processed simultaneously, the number of child server processes created at startup and the hardware infrastructure. For instance, any connection attempts over the limit of the maximum number of connections that are allowed to be processed simultaneously, will normally be queued, up to a defined number. Once a child process is freed at the end of a request, the connection will then be served.

Understanding the performance of the Apache server taking into account all the parameters that can be configured is challenging. Moreover, the complex architectural style that Apache follows makes understanding its performance a non-trivial task. Therefore, setting the many performance parameters and understanding the complex architecture of the Apache server makes it a difficult task to explore and identify its performance issues.

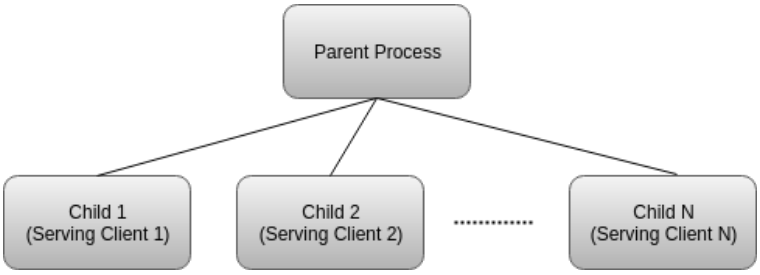


Figure 5.2 – Traditional Apache server structure [144].

5.1.1.2 MySQL server

MySQL [135] is a widely used relational database management system. MySQL provides several features like partitioned tables, commit grouping, query caching, and gathering multiple transactions from multiple connections together to increase the number of commits per second [5]. MySQL follows the client-server architecture model. There is a database server (MySQL) and arbitrarily many clients (application programs), which communicate with the server, that is, they query data, save changes, etc. Each client connection gets its thread within the server process. The connections queries execute within that single thread, which in turn resides on one core or CPU. The server *caches* threads, so that they do not need to be created and destroyed for each new connection [138]. Once a client has connected, the server verifies whether the client has privileges for each query it issues. MySQL parses queries to create an internal structure (the parse tree) and then applies a variety of optimizations. These can include rewriting the query, determining the order in which it will read tables, choosing which indexes to use, and so on. Moreover, MySQL uses a transactional

storage engine, InnoDB. This engine was designed for processing many short-lived transactions that usually complete rather than being rolled back. InnoDB has a variety of internal optimizations. These include predictive read-ahead for prefetching data from disk, an adaptive hash index that automatically builds hash indexes in memory for very fast lookups, and an insert buffer to speed up inserts.
For high availability and high throughput with low latency [5], MySQL can be deployed as a cluster consisting of one or more management nodes (`ndb_mgmd`) and data nodes (`ndbd`). The management node is responsible for storing the cluster's configuration and controlling the data nodes, whereas the data nodes are responsible for storing cluster data. After communicating with the management node, clients (MySQL clients, servers, or native APIs) connect directly to these data nodes. With MySQL cluster, there is typically no replication of data, but instead data node synchronization. For this purpose, a special data engine must be used — NDBCluster (NDB). Thus, a MySQL cluster can participate in replication with other MySQL clusters. Figure 5.3 shows a high-level overview of MySQL cluster architecture, where we have three servers: one management node, and two data nodes.

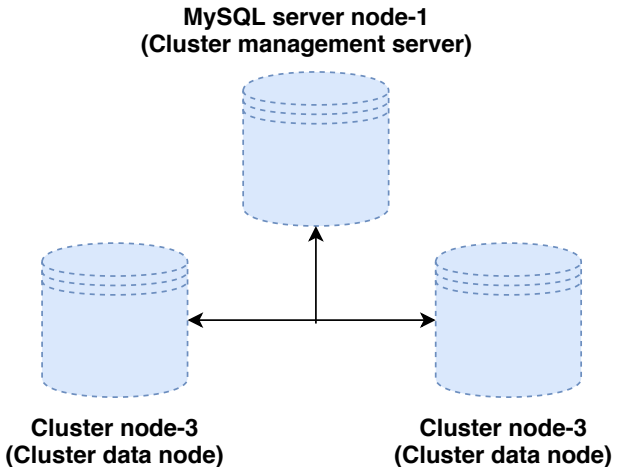


Figure 5.3 – An overview of MySQL cluster architecture

The performance of MySQL can be impacted by many factors including the maximum number of clients connections, the buffer pool size[2], etc. For example, if the data grows quickly, the buffer pool needs to over-allocate more memory otherwise the performance degrades. Understanding the performance of MySQL taking into account the complex architectural style and all the parameters that can be configured is challenging.

A high-level look at the LAMP stack order of execution shows how the elements interoperate. The process starts when the Apache Web server receives requests for web pages from a user's browser. If the request is for a PHP file, Apache passes the request to PHP, which loads the file and executes the code contained in the file. PHP also communicates with MySQL to fetch any data referenced in the code [94]. This stack is another example of a multi-tier architecture. A major difference with the case of the data processing pipeline is that communications are bi-directional in the case of the LAMP stack: The MySQL database answers to the Apache server requests, that, in its turn, answers clients requests.

The LAMP stack is widely used in production to create web applications and dynamic websites. For example, Apache Server is used by many companies [60] including eBay [4], IBM [2], VMware [8], and many more. MySQL is widely used by many companies as well [64] including Airbnb [14], Uber [19], and Netflix [7].

[2]InnoDB buffer pool is used by MySQL as cache index. It also holds row data, the adaptive hash index, the insert buffer, locks, and other internal structures [138].

5.1.2 LAMP exported metrics

In this section, we present the main performance metrics that we export at the software level for each software component in the LAMP stack. In addition to metrics at the application software level, we also export metrics at the hardware level as described in Table 3.3.

We use the prefixes defined in Table 5.1 to refer to the software components of the LAMP stack.

Component name	**Apache**	**MySQL**
Prefix	AP	MS

Table 5.1 – The prefixes of the software components names in the LAMP stack.

As the main components in the LAMP stack are Apache server and MySQL server, we focus on the performance metrics that are exported by these two components[3].

5.1.2.1 Apache performance metrics

Apache server exposes high-level metrics through its status module and logs. By consulting both of these sources, we can identify degradations and troubleshoot potential issues. When a client sends a request to an Apache server, multiple worker threads are created to process the incoming requests. The worker threads wait in a queue until an event (incoming request) has been detected. Monitoring the percentage of threads in a waiting state indicates how the server is active. On the other hand, the activity degree of the server is not enough to claim that everything is fluid in the stack. Thus, other metrics should be taken into account, such as the number of requests received, the error rate, the time required to process a request, etc. Moreover, if Apache is processing requests slowly, it could be due to other components of the stack, such as long-running queries in MySQL or it could be due to Apache itself. Furthermore, if the rate of requests is dramatically increasing, provisioning more resources might be required to handle the load. On the other hand, if the rate of requests is rapidly decreasing, it could point to problems elsewhere, e.g., the database could be crashing. Thus, metrics like request processing time, request rate, and uptime are useful to consider. The Apache error metric can be a useful indicator of underlying problems or misconfigured files. However, monitoring and analyzing the generated error code may tell that there are not enough workers available to handle the request load, or that the number of connections that could access a certain resource/page is limited [42]. Thus, we report the error rate metric based on the reported HTTP status code in Apache logs. Table 5.2 describes the main performance metrics exported by the Apache server.

5.1.2.2 MySQL performance metrics

In addition to monitoring the Apache server, if the database is running slowly, or failing to serve queries for any reason, every part of the stack that depends on that database will suffer performance problems as well. Thus performance aspects like query throughput, query execution performance, and connections are important to be tracked. As a database runs queries, the first monitoring priority should be making sure that MySQL is executing queries as expected. Metrics that capture the number of transactions or queries that have been performed per second, the number of successful queries, the number of erroneous results, and the number of slow queries, are useful to quantify how efficiently the database is doing its work. Also, to

[3]Note that in the used benchmark, an emulator is used to issue requests to the Web server. This emulator does not provide any kind of performance metrics, thus we do not consider metrics exported on the client side.

Metric	Description	Type
AP_request_rate	Represents the average number of clients requests per second.	In/Out Data
AP_bytes_per_second	Represents the amount of information being transferred in and out of the server.	In/Out Data
AP_perc_worker_idle	Represents the average percentage of processes/threads that are waiting in a queue for incoming requests.	Idle Threads
AP_request_processing	Represents the average time in microseconds required by Apache server to process clients requests.	Processing Time
AP_error_rate	Represents the number of requests that resulted in an error.	Error Rate
AP_uptime	This metric is usually measured in seconds, since the server was started. Unexpected drops in uptime can indicate unplanned outages or restarts of the server.	Uncategorized
AP_total_kbytes	Represents the average number of bytes served per second.	In/Out Data
AP_num_idle_workers	Represents the average number of processes/threads that are waiting in a queue for incoming requests.	Idle Threads
AP_total_access	Represents the average number of accesses served per second.	Uncategorized
AP_bytes_per_req	Represents the average number of bytes per request.	In/Out Data

Table 5.2 – Metrics exported by the Apache server.

understand the workload and identify potential bottlenecks, monitoring the read/write operations is helpful. Table 5.3 describes the performance metrics exported by MySQL.

Metric	Description	Type
MS_error_rate	Represents the average number of requests that resulted in an error.	Error Rate
MS_query_per_second	Represents the number of queries the server is able to process.	In/Out Data
MS_read_query	Represents the average percentage of read queries (i.e., select) that the server is processing.	In/Out Data
MS_write_query	Represents the average percentage of write queries (i.e., insert/update/delete) that the server is processing.	In/Out Data
MS_idle_threads	Represents the average percentage of threads that are sleeping.	Idle Threads
MS_slow_queries	Represents the average number of queries exceeding a configurable *long_query_timelimit*. The *long_query_timelimit* is a threshold parameter that specifies the maximum timeout in seconds that a query should not exceed.	Queue Size
MS_num_active_threads	Represents the average number of threads that are running connections. In other words, the average number of threads that are not sleeping.	Idle Threads
MS_cached_threads	Represents the average number of threads in the thread cache.	Idle Threads

Table 5.3 – Metrics exported by the MySQL server.

We have seen that many performance metrics can be exported by each software component in the LAMP stack. The question now is:*Is it possible to identify a set of key metrics that can be used as reliable indicators of performance bottlenecks in the LAMP stack?*
To reply to this question, next, we apply the proposed analytical study described in Chapter 3.

5.2 Analytical study to identify reliable indicators of performance bottlenecks for the LAMP stack

In this section, we apply the analytical study described in Chapter 3 to the LAMP stack. First, we describe the testbed and the methodology of the study. We show in detail how to apply the analytical methodology through one setup of the LAMP stack. Throughout the study on different setups we try to answer the research questions concerning the existence of *reliable* indicators of performance bottleneck and their characteristics. Then, we discuss how fine-grained the conclusions that we can draw from these indicators are. Finally, we summarize and discuss our findings over all the setups of the web stack case study.

5.2.1 Testbed and methodology

5.2.1.1 Hardware and software setup

Similarly to the case of the data processing pipeline, the experiments are performed on the Grid'5000 testbed [1] using the Lyon site. We recall that Table 3.9 shows the hardware configuration for all used clusters. We use 1 or 2 nodes for the apache server and 3 or 4 nodes for the MySQL cluster. Regarding the client side, we run each client on a separate machine (we increase the number of client machines as needed to reach the system saturation point). The system saturation point is the point where any increase in the injected load does not lead to an increase in the system throughput.

We use Debian 8 with a 3.16.0 Linux kernel, OpenJDK version 1.8.0_131, Apache 2.4.18, MySQL 5.7 and PHP 7. Regarding the data collection, metrics are collected through logs. The key Apache metrics are available through Apache's status module (*mod_status*) and the server access logs. The key MySQL metrics are available through MySQL log files (i.e., error.log and mysql-slow.log). In addition, we use *Mytop* tool [33] to collect MySQL performance data about MySQL threads, queries, etc.

We benchmark the web stack with RUBiS [88], an auction site prototype modeled after eBay [4]. RUBiS is a multi-tiered e-commerce application with 26 interaction types, such as browsing, bidding, buying, or selling items, registering users, and writing or reading comments. RUBiS provides two workload transition describing two different user behaviors: a browsing transition consisting of read-only interactions and a bidding transition, including 15% write interactions.

We consider 8 setups of the LAMP stack, described in Table 5.4. We examine 30 metrics exported on both hardware and software levels for the Apache server and the MySQL server (16 Apache metrics including resource consumption metrics and 14 MySQL metrics including resource consumption metrics).

These setups vary depending on the following factors:

- Transaction type: As described before, RUBiS provides two workload transition: read-only and read/write. We consider both types in the setups.
- Number of clients threads per machine: We test setups with two different numbers of RUBiS clients threads: 1000 and 10000.
- Number of Apache nodes: We consider setups where we vary the number of Apache servers between 1 and 2 servers.
- Number of Apache workers: The number of Apache workers represents the number of processes/threads that are responsible for processing the incoming requests. This setting has an important role as it controls the maximum total number of threads that may be launched in the Apache server [59]. We test 3 values of this setting: 150 (default value), 250, and 500.
- Number of MySQL nodes: We consider setups where we vary the number of MySQL servers between 3 and 4 servers. More precisely, we vary the number of the data nodes in the MySQL cluster between 2 and 3 nodes. This implies varying the number of replicas between 2 and 3 replicas.
- Number of MySQL buffer pool instances: Dividing the buffer pool into separate instances can improve concurrency, by reducing contention as different threads read and write to cached pages [62]. We consider 2 values of this setting: 1 instance and 16 instances.

- Size of MySQL buffer pool: The buffer pool is an area in the main memory where the InnoDB caches table and index data as it is accessed. The buffer pool permits frequently used data to be processed directly from memory, which speeds up processing. On dedicated servers, up to 80% of the physical memory is often assigned to the buffer pool [37]. We consider 2 different values of the buffer pool size: 134MB (the default value) and 8GB.

Setup	Transaction type	#Clients	Apache settings		MySQL settings		
			Num. nodes	Num. workers	Num. nodes	#Buffer pool instances	Buffer pool size
setup-0	Read/write	1000	1	150	3	1	134MB
setup-1	Read/write	1000	2	150	3	1	134MB
setup-2	Read/write	10000	2	150	3	1	134MB
setup-3	Read/write	1000	1	150	4	1	134MB
setup-4	Read/write	1000	1	150	3	16	8GB
setup-5	Read/write	1000	1	250	3	1	134MB
setup-6	Read/write	1000	1	500	3	1	134MB
setup-7	Read	1000	1	150	3	1	134MB

Table 5.4 – Description of the different setups for the LAMP stack.

5.2.1.2 Methodology

We apply the same analytical study as described in Chapter 3. The main performance metric we are interested in is the *global throughput* of the system i.e., the amount of work that the system is able to handle per time unit. In the case of a Web stack, the work corresponds to the processing of request-response exchanges. We study the performance of a pair of interacting components A and B where A represents the client side and B represents the server side. Note that in the case of the LAMP stack, component B is always composed of two software components: Apache server and MySQL database as shown in Figure 5.4.

To identify the limiting component in the system, we strive to identify the point of *peak* throughput, where, while increasing the number of clients, the component limiting the performance of the system is shifting from one side to the other. Before this point, we consider that the component A is the *bottleneck* component, meaning that performance could be improved by increasing the intensity of the load injected by A. Conversely, at this point *and beyond*, we consider that the component B is the *bottleneck* component, meaning that performance can only be improved by provisioning more hardware resources for B (or reconfiguring B). On the other hand, there might be a *gray zone* where it is ambiguous to identify if the client (i.e., component A) is still the bottleneck or if the bottleneck shifts completely to the server side (i.e., component B).

We insist that B (i.e., the server) is a bottleneck when B reaches its maximum capacity in terms of *global throughput* within the given hardware resource budget. Note that we use the notion of A (i.e., the client side) is a bottleneck to refer to the fact that the server has not reached its maximum capacity with the current load injection.

5.2.2 Detailed analysis of one setup

In this section, we illustrate the analytical approach on one setup of the LAMP stack case study. The goal here is to answer the question concerning the existence of reliable indicators of performance bottlenecks in a given setup of the LAMP stack. We consider `setup-0`, described in Table 5.4. We measure the global throughput of the system while increasing the number of clients. Figure 5.5 shows the system throughput as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the system throughput in req/s. We compute the *peak throughput range* which is: $[503 \times 0.95;\quad 503] =$

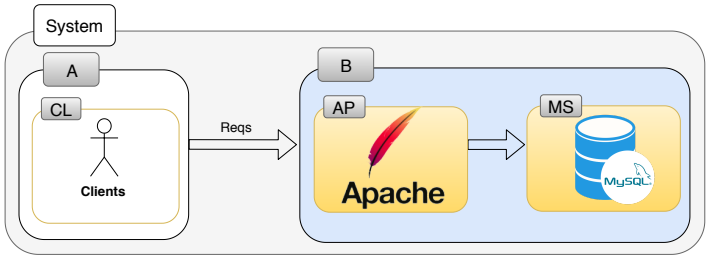


Figure 5.4 – Illustration of a Web stack comprising a pair of interacting components A and B, where B is a complex component composed of Apache server and MySQL database.

[478; 494]. $Th_{0.95} = 478$ req/s corresponds to 6 clients. Based on our definition of the bottleneck, this implies that the client is the bottleneck until we reach 6 clients injecting load. At 6 clients and beyond the bottleneck shifts to be on the server side (the bottleneck might be on the Apache server side or in the MySQL server side or on both).

Note that the LAMP stack behaves differently from the data processing pipeline in configurations where the server has not yet reached its maximum capacity. Namely, even before that point adding more hardware resources on the server side always leads to an improvement in the throughput. The reason is that, since clients are waiting for answers from the server in the case of the LAMP stack, improving the latency on the server side also has a direct positive impact on the global throughput of the system. As a consequence, defining a gray zone where adding more resources on the server side starts improving the performance is meaningless in this case. Hence, in the following analysis, we consider only two cases. Either there are not enough clients (until $Th_{0.95}$) or the server has reached its maximum capacity (after $Th_{0.95}$).

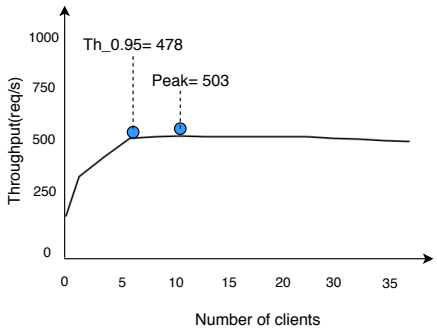


Figure 5.5 – Bottleneck identification for the LAMP stack with setup-0.

In the following sections, we present and analyze the evolution of the metrics of both the Apache server and the MySQL server. We aim at classifying these metrics into pattern categories as described in Section 3.3.2.6. In all the figures presenting the evolution of metrics, the green points represent cases where the bottleneck is on the client side and the red points represent the bottleneck being on the server side.

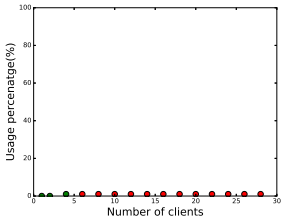
5.2.2.1 Evolution of Apache server metrics

First, we present the evolution of resource consumption metrics. Then, we present the evolution of the software level metrics for the Apache server.

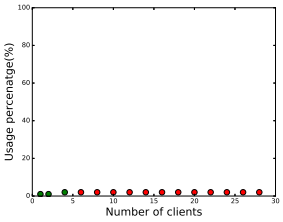
Figure 5.6 shows the evolution of resource consumption metrics for the Apache server as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. From Figure 5.6, we observe that resource consumption metrics are never *reliable* indicators of performance

bottlenecks in the given setup of the LAMP stack. The value of these metrics is constant, i.e., there are no changes neither in the metrics values nor in their behavior when the bottleneck shifts from one side to the other.

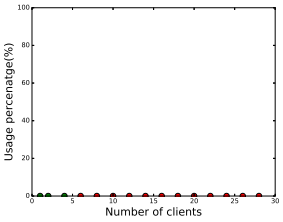
Figure 5.7 shows the evolution of the Apache server software level metrics as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Figure 5.7 (e) shows a change in the metric values and in its behavior when the bottleneck shifts from the client side to the server side. Thus, we consider this metric as a *reliable* indicator. Figure 5.7 (d) acts as a *misleading* indicator as the change in the metrics values does not occur at the point where the bottleneck shifts from the client side to the server side. The rest of the metrics act either as *partially reliable* or *not reliable* indicators of performance bottleneck.



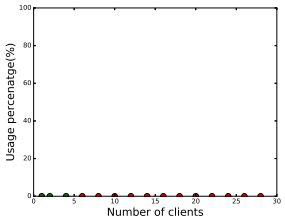
(a) Avg. CPU usage(%).



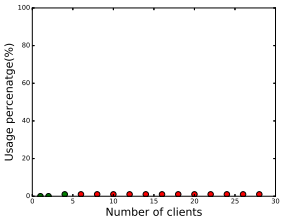
(b) Avg. memory usage(%).



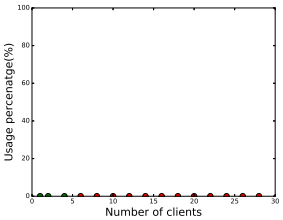
(c) Avg. disk read usage(%).



(d) Avg. disk write usage(%).

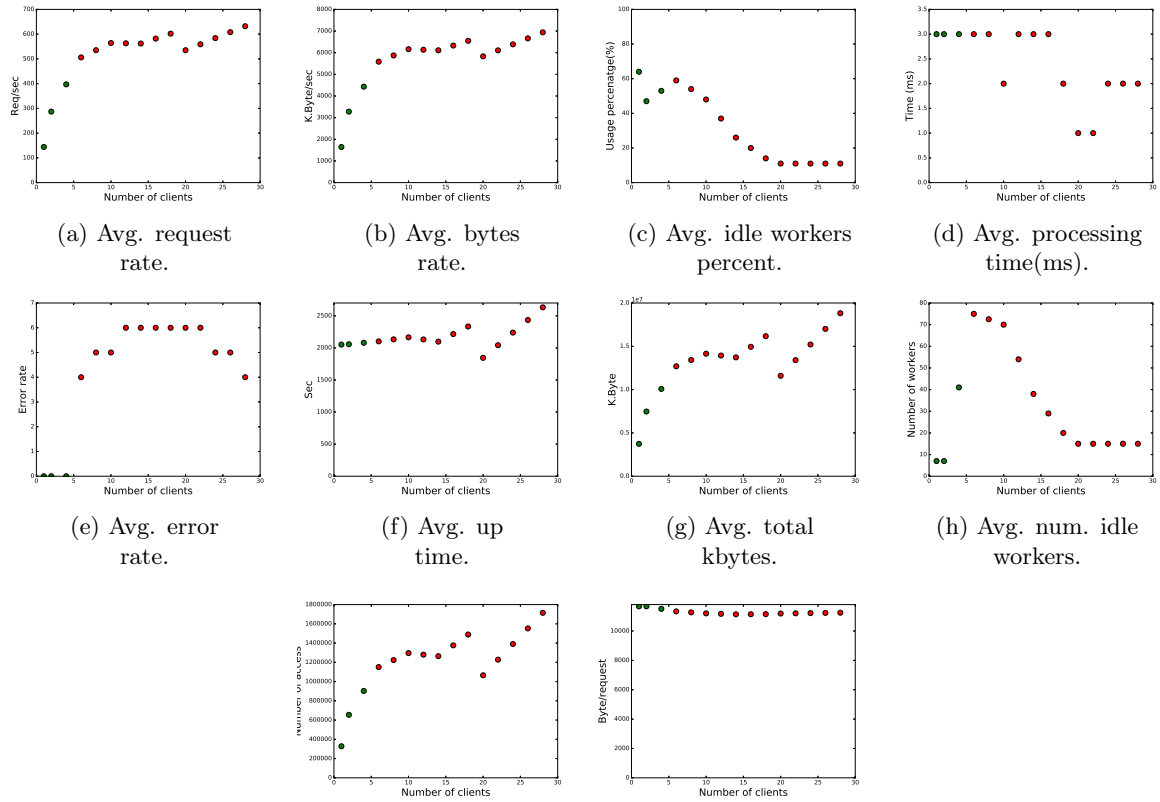


(e) Avg. network sent usage(%).



(f) Avg. network receive usage(%).

Figure 5.6 – Evolution of Apache resource consumption metrics for setup-0.



(a) Avg. request rate. (b) Avg. bytes rate. (c) Avg. idle workers percent. (d) Avg. processing time(ms).

(e) Avg. error rate. (f) Avg. up time. (g) Avg. total kbytes. (h) Avg. num. idle workers.

(i) Avg. total accesses. (j) Avg. bytes per request.

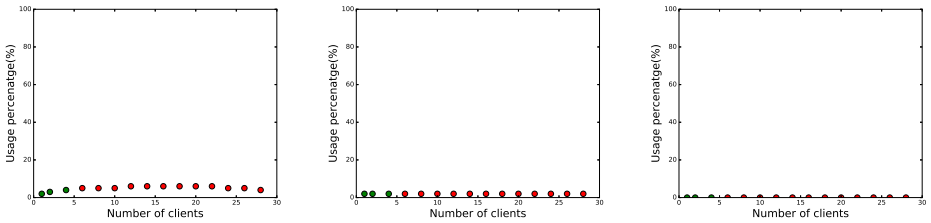
Figure 5.7 – Evolution of Apache software level metrics for setup-0.

5.2.2.2 Evolution of MySQL metrics

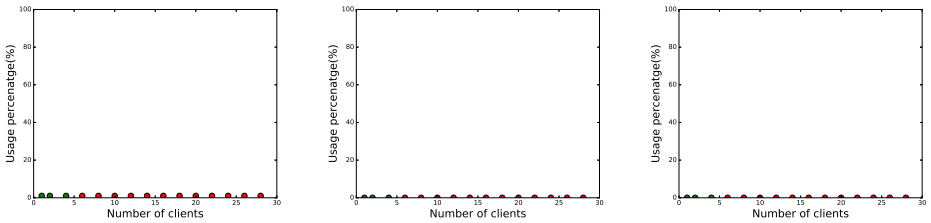
In this section, we show the evolution of resource consumption metrics and the evolution of the software level metrics for the MySQL server.

Figure 5.8 shows the evolution of resource consumption metrics for MySQL server as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Similarly to the Apache server case, resource consumption metrics are never *reliable* indicators of performance bottlenecks.

Figure 5.9 shows the evolution of the software level metrics of MySQL server as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the metric values. Based on Figure 5.9, we observe that most MySQL metrics are either *not reliable* or *misleading* indicators of performance bottleneck. The only exception is Figure 5.9 (h) where the metric evolution follows constant-linear pattern. Moreover, the metric changes its behavior at the point where the bottleneck shifts from the client side to the server side while similar values are observed when the bottleneck is on the client side or on the server side. Consequently, this metric is considered as a *partially reliable* indicator.

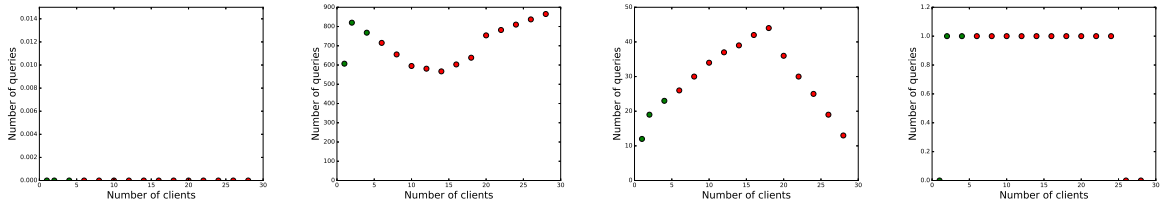


(a) Avg. CPU usage(%). (b) Avg. memory usage(%). (c) Avg. disk read usage(%).

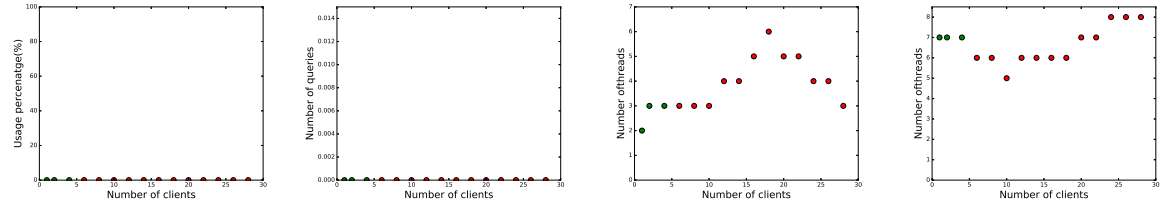


(d) Avg. disk write usage(%). (e) Avg. network sent usage(%). (f) Avg. network receive usage(%).

Figure 5.8 – Evolution of MySQL resource consumption metrics for setup-0.



(a) Avg. error rate. (b) Avg. query rate. (c) Avg. percent read query. (d) Avg. percent write query.



(e) Avg. idle threads. (f) Avg. num. slow queries. (g) Avg. num. active threads. (h) Avg. num. cached threads.

Figure 5.9 – Evolution of MySQL software level metrics for setup-0.

5.2.3 Are resource consumption metrics sufficient?

To answer this question, we check the categories that resource consumption metrics belong to for both Apache and MySQL servers for all setups of the LAMP stack. Figures 5.10 and 5.11 show the categories of resource consumption metrics for Apache and MySQL servers respectively. We observe that resource consumption metrics are never *reliable* indicators of performance bottlenecks for any studied setup of the LAMP stack. Thus, similarly to what observed in the data processing pipeline we cannot rely on resource consumption metrics to identify where a bottleneck lies in the LAMP stack.

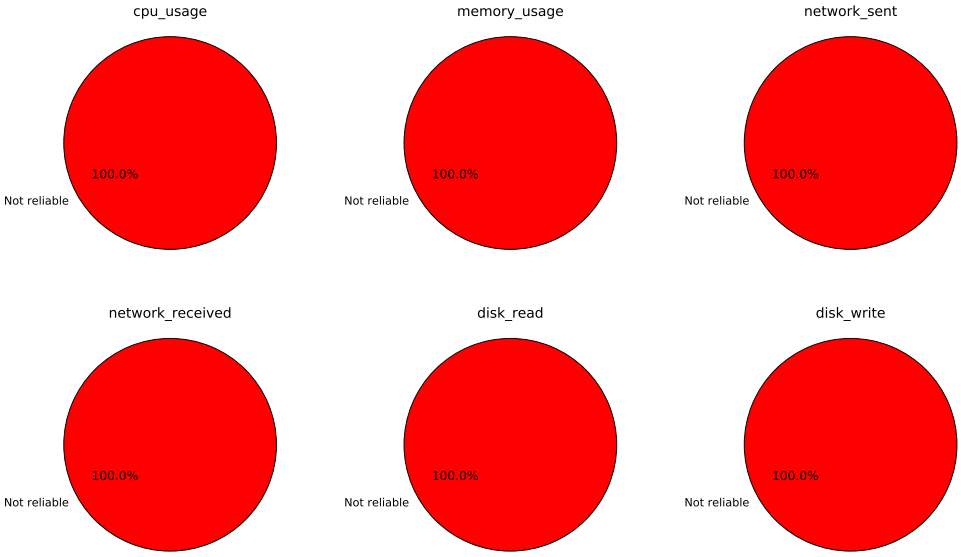


Figure 5.10 – The pattern category (in percent) that Apache server resource consumption metrics follow for all setups of the LAMP stack.

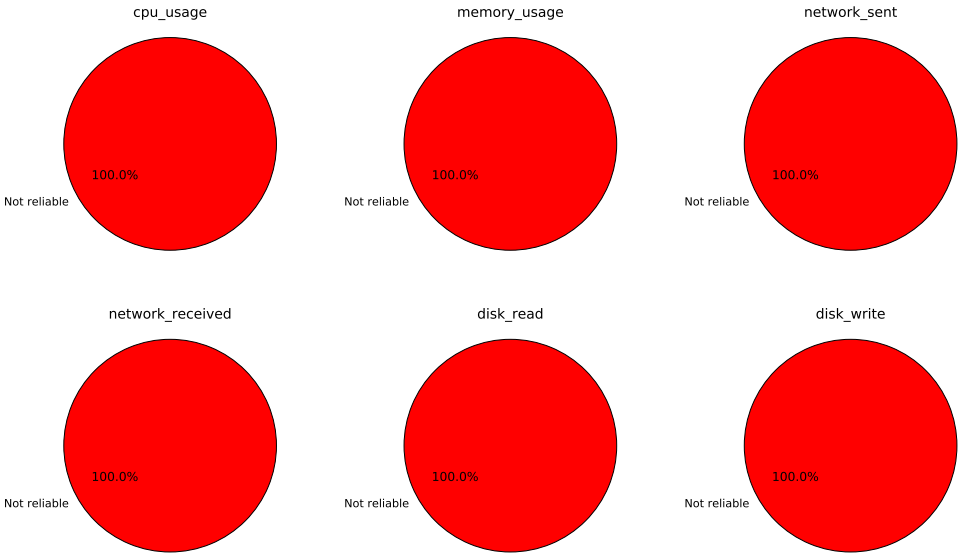


Figure 5.11 – The pattern category (in percent) that MySQL resource consumption metrics follow for all setups of the LAMP stack.

5.2.4 Are there *reliable* indicators of performance bottleneck in the LAMP stack?

Similarly to the data processing pipeline case study, in this section, we strive to answer the question concerning the existence of *reliable* indicators of performance bottleneck for the setups of the LAMP stack. We apply the analytical study on all setups of the LAMP stack described in Table 5.4. Table 5.5 summarizes the output of the study. From Table 5.5, we observe that:

- For a given setup of the LAMP stack, there is always at least one *reliable* metric that we can rely on to identify the performance bottleneck in the system.
- There are always Apache metrics that act as *reliable* indicators of performance bottleneck for any given setup. Also, most Apache metrics act as *reliable* indicators in at least one setup (except for `AP_upime` metric which never acts as reliable for any setup). On the other hand, only few MySQL metrics act as *reliable* indicators in at least one setup (e.g., `MS_slow_queries`, and `MS_idle_threads` are never *reliable* indicators for any test setup).
- Relying on the `AP_error_rate` metric is sufficient to cover almost all the setups of the LAMP stack as it acts as a *reliable* indicators for 7/8 setups (it acts as a *partially reliable* for **setup-4**).
- In order to cover all setups of the LAMP stack, Figure 5.12 shows a possible combination of one Apache metric, `AP_error_rate`, with one of these two metrics: `AP_bytes_per_req` and `MS_cached_threads`.
- Eliminating the misleading metrics from any combination to cover most setups of the LAMP stack, we find that the `AP_error_rate` metric never acts as a *misleading* indicator. Thus, we can rely on this metric to cover 7/8 setups. On the other hand, we observe that all other metrics that behave as *reliable* indicators for some setups, behave as *misleading* indicators for other setups.

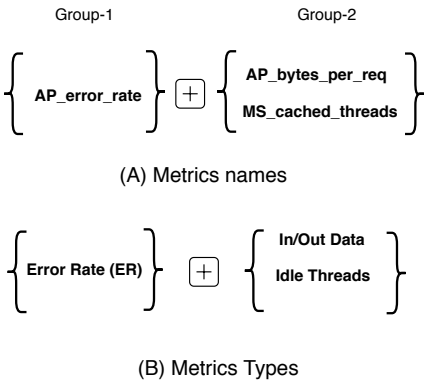


Figure 5.12 – A possible combination of *reliable* metrics for the LAMP stack case study.

5.2.5 What are the characteristics of performance bottleneck indicators?

Depending on the tested setup of the LAMP stack, the category to which a metric belongs may change. Figure 5.13 shows pie charts representing, for each Apache software level metric, the category (as a percentage) that the metric belongs to depending on the setup. Figure 5.14 shows the same information, for each MySQL software level metric (as a percentage), the category that the metric belongs to depending on the setup.

From both Figures 5.13 and 5.14, we observe that:

Metrics	Setups							
	0	1	2	3	4	5	6	7
Apache metrics								
cpu_usage	NR	NR	NR	NR	NR	NR	NR	NR
memory_usage	NR	NR	NR	NR	NR	NR	NR	NR
network_sent	NR	NR	NR	NR	NR	NR	NR	NR
network_received	NR	NR	NR	NR	NR	NR	NR	NR
disk_read	NR	NR	NR	NR	NR	NR	NR	NR
disk_write	NR	NR	NR	NR	NR	NR	NR	NR
AP_request_rate	PR	M	M	**R**	NR	**R**	**R**	**R**
AP_bytes_per_second	PR	M	M	**R**	NR	**R**	**R**	**R**
AP_perc_worker_idle	PR	NR	PR	PR	PR	PR	PR	PR
AP_request_processing	M	NR	NR	**R**	PR	M	M	NR
AP_error_rate	**R**	**R**	**R**	**R**	PR	**R**	**R**	**R**
AP_uptime	NR	NR	NR	NR	M	NR	M	M
AP_total_kbytes	NR	NR	NR	NR	NR	M	**R**	**R**
AP_num_Idle_workers	PR	M	M	PR	PR	**R**	**R**	NR
AP_total_access	NR	NR	NR	NR	NR	M	**R**	M
AP_bytes_per_req	NR	NR	NR	M	**R**	**R**	**R**	**R**
MySQL metrics								
cpu_usage	NR	NR	NR	NR	NR	NR	NR	NR
memory_usage	NR	NR	NR	NR	NR	NR	NR	NR
network_sent	NR	NR	NR	NR	NR	NR	NR	NR
network_received	NR	NR	NR	NR	NR	NR	NR	NR
disk_read	NR	NR	NR	NR	NR	NR	NR	NR
disk_write	NR	NR	NR	NR	NR	NR	NR	NR
MS_error_rate	NR	PR	NR	NR	NR	NR	NR	NR
MS_query_per_second	NR	M	M	PR	PR	PR	PR	**R**
MS_read_query	M	M	M	NR	NR	NR	NR	NR
MS_write_query	M	PR	NR	NR	M	M	M	NR
MS_idle_threads	M	NR	NR	NR	NR	NR	NR	NR
MS_slow_queries	NR	PR	NR	NR	NR	NR	NR	NR
MS_num_active_threads	M	M	M	**R**	M	M	M	NR
MS_cached_threads	PR	**R**	**R**	**R**	**R**	PR	M	**R**

Table 5.5 – Analytical study results for all setups of the LAMP stack (R: reliable, PR: partially reliable, M: misleading, and NR: not reliable).

- For the Apache server, the metrics `AP_request_rate`, `AP_bytes_per_second`, `AP_error_rate`, and `AP_bytes_per_request` act as *reliable* indicators of performance bottleneck for a majority of the setups. These metrics belong to the two following types (see Table 3.2): *In/Out Data (IOD)* and *Error Rate (ER)*. The `AP_perc_worker_idle` metric acts as *partially reliable* indicator for most of the setups. The rest of the Apache server metrics are either *not reliable* or *misleading* categories for most of the setups.
- For the MySQL server, `MS_cached_threads` is the only metric that acts as a *reliable* indicator for a majority of the studied setups. The `MS_error_rate` metric is never *reliable* indicator of performance bottleneck. The `MS_query_per_second` acts as a *partially reliable* indicator for most of the setups. The categories of the rest of the metrics vary between *not reliable* and *misleading* for most of the setups.

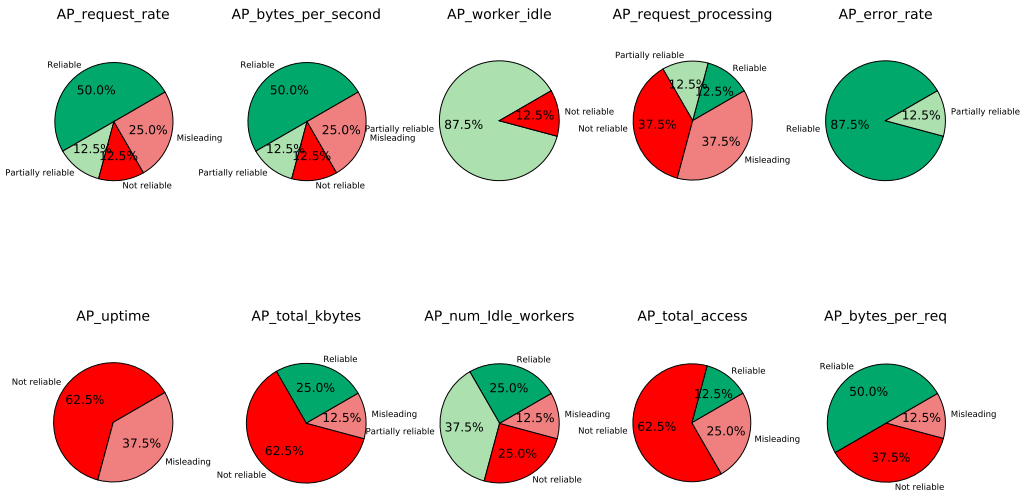


Figure 5.13 – The pattern category (in percent) that Apache software level metrics follow for all setups of the LAMP stack.

5.2.6 How fine-grained are the conclusions that we can draw from these metrics?

In this section, we strive to check if relying on the *reliable* indicators allows us not only identifying where the bottleneck is (client or server) but also, in cases where the server is the bottleneck, identifying which component is limiting (i.e., Apache server or MySQL). To find the limiting component in the system, we perform the same kind of experiments as we did for the data processing pipeline: for each component at a time on the server side, we add more hardware resources and, we measure the improvement in the global throughput of the system. We consider that the component that improves the throughput the more by adding more hardware resources, is the limiting component in the system. To illustrate the methodology, we consider **setup-0**. We add one node to the Apache server and one node to MySQL at a time. We measure the system throughput in both cases. Figures 5.15 (a) and (b) show the system throughput as a function of the number of clients. The X-axis represents the number of clients and the Y-axis represents the system throughput measured by requests per second. Figure 5.15 (a) shows that adding one more Apache server

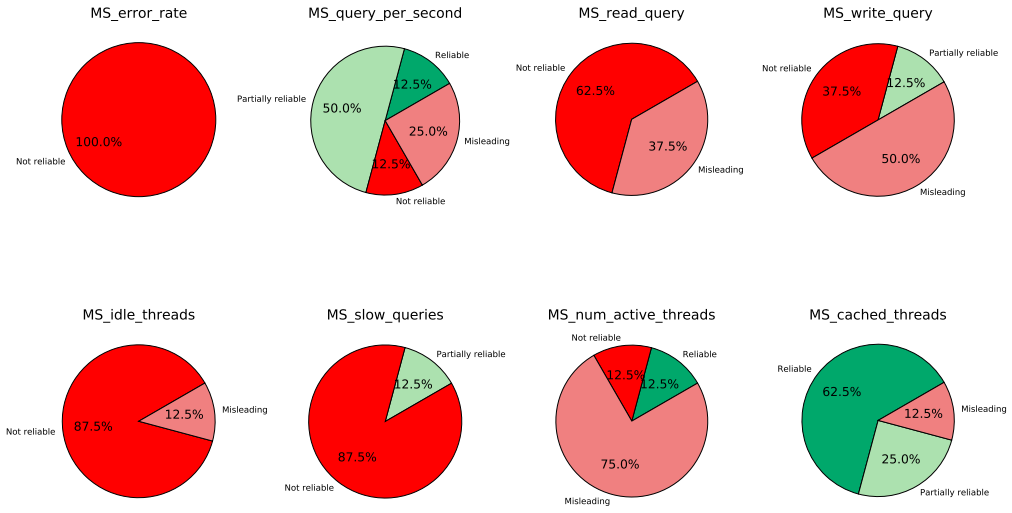
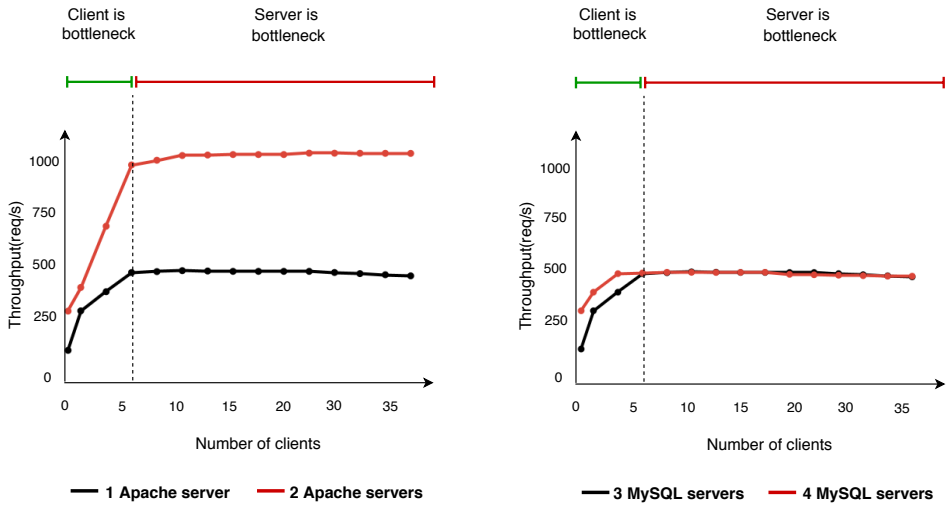


Figure 5.14 – The pattern category (in percent) that MySQL software level metrics follow for all setups of the LAMP stack.

improves the system throughput by two times compared with the one-apache-server setup. Figure 5.15 (b) shows that adding one more server to the MySQL cluster does not improve the system throughput. Thus, based on the two Figures 5.15 (a) and (b), we conclude that the Apache server is the bottleneck in this given setup.

Now, the question is: Can the *reliable* metrics pinpoint that the Apache server is limiting the performance in `setup-0`? Table 5.5 shows that `AP_error_rate` is the only *reliable* indicator for **setup-0** as it changes its behavior and values when the bottleneck shifts to the server side while other metrics do not. This might be a hint that the Apache server is the bottleneck in the given setup.

On the other setups of the LAMP stack, the Apache server component is also always the bottleneck. Based on the results presented in Table 5.5, we see that there are several cases where only Apache server metrics act as reliable indicators of performance bottleneck. On the other hand, there are setups where MySQL metrics also act reliable indicators, and where it would be more difficult to conclude that the Apache server is the limiting component. Setups, where MySQL is the limiting component, would have to be studied to see if they significantly differ from the setups studied here.



(a) LAMP stack throughput with 2 Apache nodes.

(b) LAMP stack throughput with 4 MySQL nodes.

Figure 5.15 – LAMP stack throughput when provisioning more hardware resources.

5.2.7 Summary of the reliable metrics types for the data processing pipeline and the Web stack

Based on our analytical and experimental study on the LAMP stack with different setups, we summarize our findings as follows:

- Resource consumption metrics are never *reliable* indicators of performance bottleneck in any of the tested setups of the LAMP stack.
- For each tested setup, there are always some metrics that are *reliable* indicators of performance bottleneck in the LAMP stack.
- Relying on the `AP_error_rate` metric is sufficient to cover most of the cases of the LAMP stack as it acts as a *reliable* indicator for 7/8 setups.
- In order to cover all setups of the LAMP stack, Figure 5.12 shows a possible combination of one Apache metric (i.e., `AP_error_rate`) with one of these two metrics: `AP_bytes_per_req` and `MS_cached_threads`. This implies that a combination of different metrics types is necessary to identify performance bottleneck in the LAMP stack as illustrated by Table 5.6.
- Comparing the types of the reliable metrics for the data processing pipeline and for the LAMP stack, we observe that **Idle Threads (ITD)**, **Error Rate (ER)**, and **In/Out Data (IOD)** are common types for the two systems as Table 5.7 shows.

To summarize, we can say that these main results obtained with the LAMP stack are consistent with the results obtained in Chapter 3 when analyzing the data processing pipeline.

Metric Type	Web stack	
	Apache server	MySQL
Resource Consumption (RC)		
Idle Threads (ITD)		✓
Error Rate (ER)	✓	
Queue Waiting Time (QWT)		
Queue Size (QS)		
Latency (LY)		
Processing Time (PT)		
In/Out Data (IOD)	✓	
Uncategorized (UC)		

Table 5.6 – Summary of reliable metrics types that can be combined to identify performance bottlenecks for the LAMP stack.

Metric Type	Full data pipeline	Web stack
Resource Consumption (RC)		
Idle Threads (ITD)	✓	✓
Error Rate (ER)	✓	✓
Queue Waiting Time (QWT)	✓	
Queue Size (QS)	✓	
Latency (LY)		
Processing Time (PT)	✓	
In/Out Data (IOD)	✓	✓
Uncategorized (UC)		

Table 5.7 – Summary of reliable metrics types for the data processing pipeline and the LAMP stack.

5.3 Learning-based approach on LAMP

In this section, we apply our proposed learning-based approach to the LAMP stack. Our goal is to verify whether the main results obtained in Chapter 4 are also valid with this stack.

We study what metrics should be given as input to a classification algorithm to be able to accurately identify performance bottlenecks. To this end, we compare the accuracy obtained when building models using different subsets of metrics including a subset that corresponds to the metrics used by some related works [109, 150, 161]. The sets of metrics we consider are:

- **All**: It represents models that use *all* metrics exported by system components on both hardware and software levels.
- **Recommended**: It represents models that use only *recommended* metrics. By recommended metrics, we refer to metrics identified as reliable indicators in Section 5.2. The set of *recommended* metrics includes: `AP_error_rate`, `AP_bytes_per_req`, `AP_request_rate`, `AP_bytes_per_second`, `AP_total_access`, `AP_total_kBytes`, and `MS_cached_threads`.
- **Server-side-only**: It represents models that use only metrics exported on the software level by components on the server side (resource consumption metrics are excluded in this subset).
- **Resource consumption (RC)**: It represents models that use only resource consumption metrics (i.e., CPU, memory, network, and disk IO).

- **In/Out data**: It represents models that use only metrics belong to the **In/Out Data (IOD)** type. This subset of metrics includes: `AP_request_rate`, `AP_bytes_per_second`, `AP_total_kbytes`, `AP_bytes_per_req`, `MS_query_per_second`, `MS_read_query`, and `MS_write_query`.
- **Waiting time (WT)**: It represents models that use only waiting time metrics. This subset includes: `AP_perc_worker_idle`, `AP_num_idle_workers`, `MS_total_connected_threads`, `MS_active_threads`, and `MS_cached_threads`.

Note that for the LAMP stack, we choose to study the subset of metrics corresponding to the **In/Out data** type, instead of the **response time** type in the data processing pipeline, because of the significant number of the exported metrics in the LAMP stack that fall into this type.

5.3.1 Methodology

We use the same methodology as described in Chapter 4 for the data processing pipeline case study. We test 4 different classifiers on a large set of metrics for the different setups of the LAMP stack. The data set is composed of the exported metrics on both the hardware level (i.e., resource consumptions metrics) and the software level (i.e., Apache metrics and MySQL metrics). To validate the accuracy of a model, we use the *Leave-p-out* cross-validation schema, as we did in Chapter 4. For a detailed description of the methodology for each tested case, we refer the reader to the descriptions provided in Sections 4.3.1 and 4.3.2.

5.3.2 What kind of metrics can we use to build an accurate learning model?

In this section, we aim at answering the question of whether machine learning techniques applied to metrics can identify performance bottleneck in the LAMP stack. If so, we would also like to know what kind of metrics are more reliable to build an accurate learning model. We follow the same methodology described for the data processing pipeline use case (Section 4.3.2.1). Figure 5.16 shows the average accuracy for the different models with different classifiers for the LAMP use case. The X-axis represents different metrics selections with the 4 different classifiers. The Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. From Figure 5.16, we observe that:

- Only models based on *recommended*, *server-side-only*, and *In/Out data* metrics manage to achieve an accuracy (i.e., up to 88%) that is higher than the baseline (i.e., 78%).
- Models based on *recommended* metrics achieve the best accuracy with the RF classifier (accuracy of 88%).
- Models based on *resource consumption* metrics achieve the worst accuracy for all classifiers compared with other models.

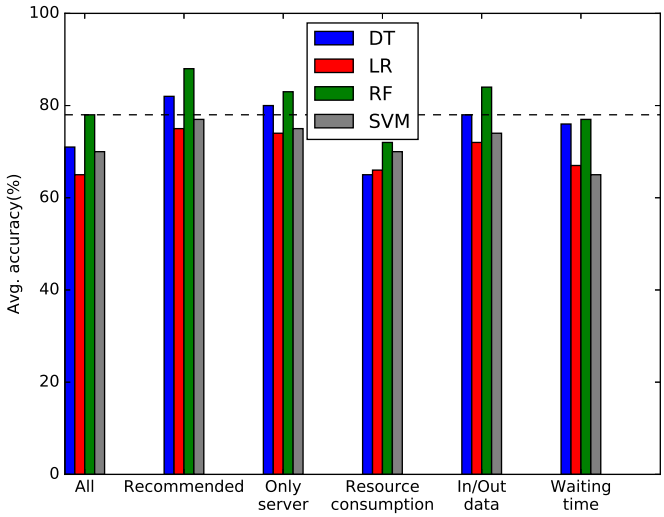


Figure 5.16 – Average accuracy when predicting for a new setup for different metrics selections and different classifiers for the LAMP stack use case.

We show the accuracy distribution for the different models in Figure 5.17. We only show the accuracy distribution with the Random Forest classifier as it shows the best accuracy for most models. Figure 5.17 shows the models accuracy distribution of the different models. The X-axis represents the different metrics selections models and the Y-axis represents the model accuracy distribution. From Figure 5.17, we can observe that only models based on *recommended* metrics have a minimum accuracy which is above the baseline (the baseline accuracy is 78%).

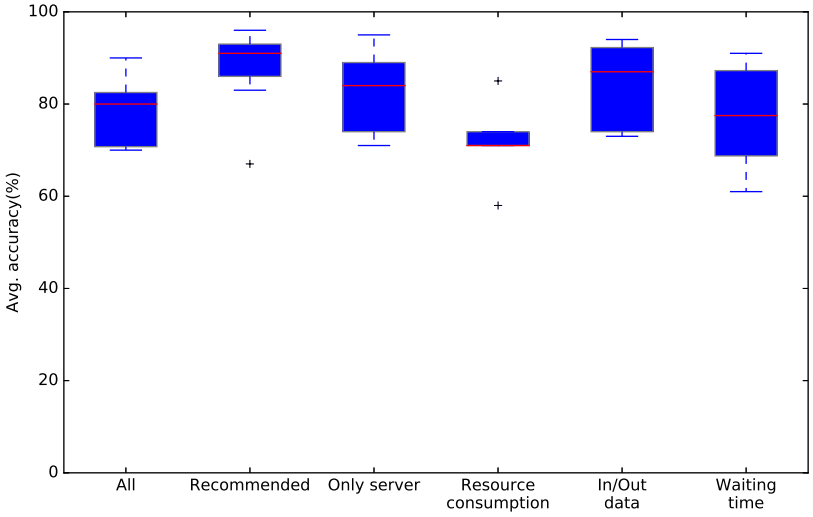


Figure 5.17 – Model accuracy distribution using *Leave-p-out* schema for the LAMP stack with RF classifier, when predicting on a new setup.

5.3.3 What are the important features of the learning model?

In this section, we show the set of features (metrics) that have high importance in building training models based on *all* metrics exported by the Web stack components (i.e., Apache server and MySQL) on both the hardware and the software levels.

Figure 5.18 shows the average importance for the top 10 features (metrics) that participate in constructing models that use *all* metrics with the Random Forest classifier. The X-axis represents how important the feature is and the Y-axis represents the model features (metrics). From Figure 5.18, we observe that:

- Metrics that belong to the **Error Rate (ER)** (e.g., `AP_error_rate`), **In/Out Data (IOD)** (e.g., `MS_query_per_second`), and **Idle Threads (IDT)** (e.g., `AP_num_idle_workers`) types are of high level of importance to the learning model.
- Metrics exported by Apache server are more important to the learning model than metrics exported by MySQL.

These results are consistent with the conclusions of our analysis presented in Section 5.2. We can also notice that the `AP_error_rate` metric, that is the most reliable indicator according to Table 5.5, appears to be a very important feature for the model in Figure 5.18.

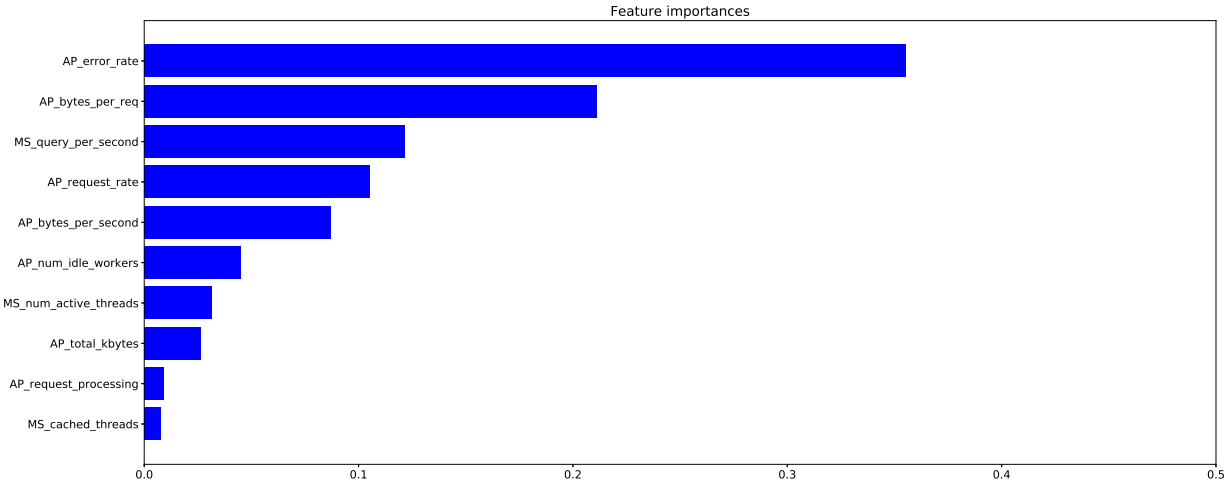
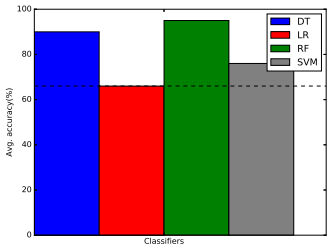


Figure 5.18 – Average importance of the proposed model features with the RF classifier for the LAMP stack.

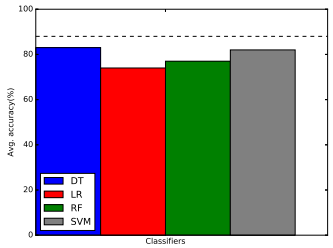
5.3.4 Are there setups that are more prone to wrong predictions?

In this section, we study if some setups are more prone to wrong predictions. To do so, we apply the same methodology as described in Section 4.3.2.3. We use models based on *recommended* metrics as they show better accuracy compared with other models. Figure 5.19 shows the accuracy for each setup when it is used as a test set. The X-axis represents the different classifiers and the Y-axis represents the model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. Note that for `setup-3` and `setup-4`, the baseline does not appear as the server is always the bottleneck, i.e., the baseline achieves an accuracy of 100%. From Figure 5.19, we notice that:

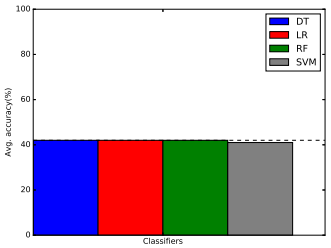
- For most setups, there is at least one classifier that achieves better accuracy than the baseline.
- The three setups for which the accuracy is lower than the baseline are: **setup-1**, **setup-3**, and **setup-4**. For **setup-3** and **setup-4**, it is explained by the fact that the baseline reaches 100% of accuracy. For **setup-1**, it might be explained by the fact that several metrics behave as misleading for this setup according to Table 5.5.



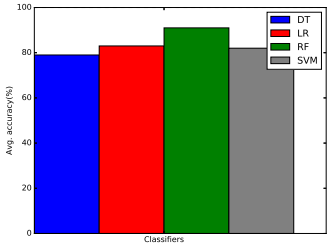
(a) Accuracy for testing setup-0.



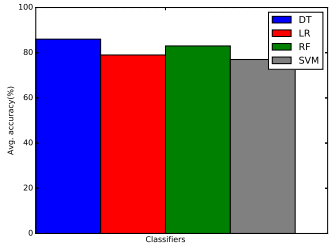
(b) Accuracy for testing setup-1.



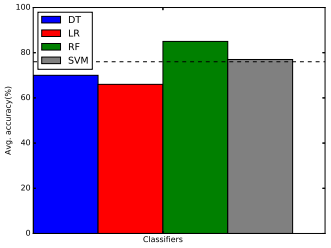
(c) Accuracy for testing setup-2.



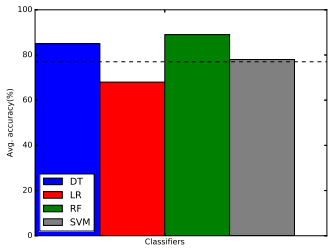
(d) Accuracy for testing setup-3.



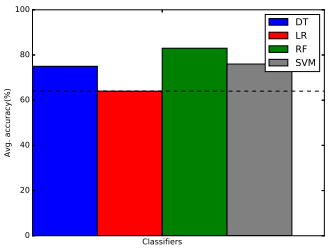
(e) Accuracy for testing setup-4.



(f) Accuracy for testing setup-5.



(g) Acuracy for testing setup-6.



(h) Accuracy for testing setup-7.

Figure 5.19 – Accuracy for all setups of the LAMP stack when using the recommended metrics with different classifiers..

5.3.5 Training and testing on the same setup but with different numbers of clients

In this section, we test the capacity of models to generalize to a new number of clients. In this experiment, the tested setup also appears in the training set but with runs involving a different number of clients. We apply the same methodology as described in Section 4.3.2.4.

Figure 5.20 shows the average accuracy for overall tested setups of the LAMP cluster. The X-axis represents the models with different metrics selections and the Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck. From Figure 5.20, we observe that:

- Models based on *recommended*, *server-side-only*, and *In/out data* metrics are still the more reliable for most classifiers.
- The best performance is achieved with models that use *server-side-only* metrics with the DT classifier (accuracy of 91%).
- More generally, in cases where the server configuration is already known but with a different number of clients, an accuracy above 90% can be achieved.
- Models based on *resource consumption* metrics perform as the worst for most classifiers.

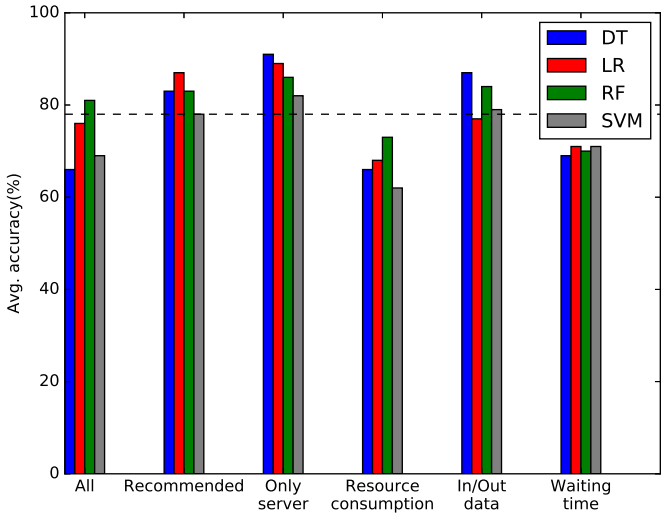


Figure 5.20 – Average accuracy when predicting for a new number of clients for different metrics selections and different classifiers for the LAMP stack.

5.3.6 Training and testing on different numbers of clients for different setups

In this section, we examine a hard case where we test with a number of clients and a setup that never appear in the training set. The goal here is to check if our model can learn to predict correctly even when it comes to a case with a new number of clients for a new setup. We apply the same methodology as described for the data processing pipeline in Section 4.3.2.5.

Figure 5.21 shows the average accuracy for testing a different number of clients for the different metrics selections models. The X-axis represents the different metrics selections models with 4 different classifiers and the Y-axis represents the average model accuracy. The horizontal dashed line represents the baseline which is the accuracy obtained when always predicting that the server is the bottleneck.
From Figure 5.21, we observe that:

- Models based on *all* and *recommended* metrics achieve the best accuracy with the DT and LR classifiers respectively (accuracy up to 90%).
- Models based on *recommended* metrics are still the most reliable for most classifiers. This confirms that combining metrics of different kinds can help in identifying performance bottlenecks in the LAMP stack.

- Models based on *server-side-only* and *in/out data* metrics achieve similar results.
- Models based on *resource consumption* and *waiting time* metrics achieve the worst accuracy for all classifiers compared with the baseline (i.e., 78%).

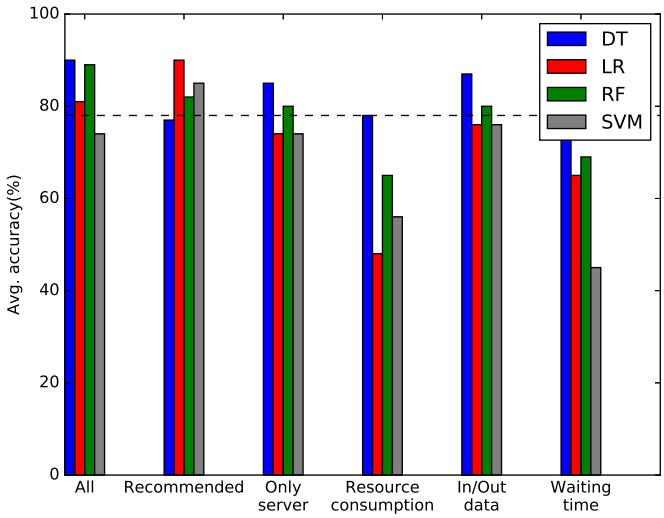


Figure 5.21 – Average accuracy when predicting for a new number of clients and a new setup for different metrics selections and different classifiers for the LAMP stack use case.

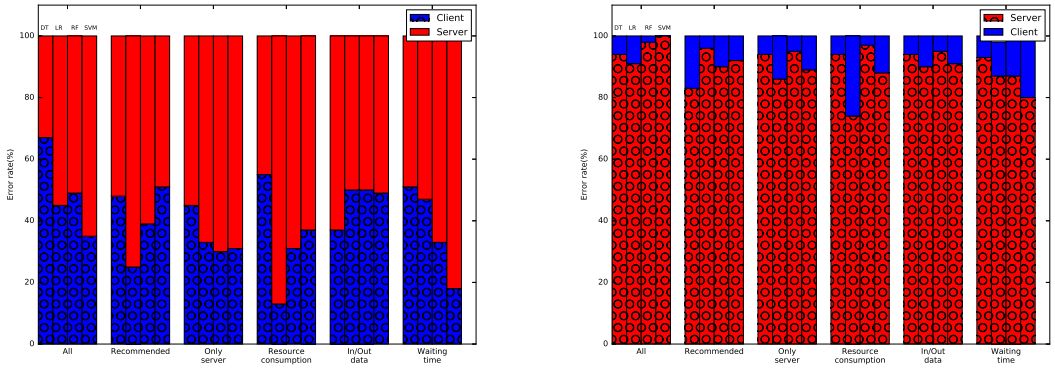
To better understand for this case, where the created models make mistakes in their prediction, we analyze the predictions that are made and compare them to the real class of the tested data. We recall that the class of a data corresponds to the bottleneck in the studied configurations, that is, `client` or `server`. For each class, we present the error rate.

Figures 5.22 (a) and (b) show, for all models with 4 classifiers, the average error rate for the `client` and the `server` classes respectively. The X-axis represents the different metrics selections and the Y-axis represents the average error rate. Each bar in the figure represents a classifier. Each bar also shows 2 areas where the bottom of the bar represents the correct prediction and the other part represents the errors that the model predicts in the other class. In Figure 5.22 (a), the blue bottom part of each bar represents the average correct prediction of the client being a bottleneck and the red part represents the average error that the model makes by predicting the server being a bottleneck instead of the client. For Figure 5.22 (b), the correct predictions are in red.

From Figures 5.22 (a) and (b), we observe that:

- Models based on *all* metrics achieve the best accuracy with the DT classifier for client predictions.
- In the case of predicting a server being a bottleneck, all models achieve good prediction for most classifiers.
- In general models based on *recommended* achieve good accuracy for client and server predictions.

On Figure 5.22 (b), we can observe that we almost never predict `client` instead of `server`. In Figure 5.22 (a), we observe that only one model based on *all* metrics predicts `server` instead of `client` in less than 35% of the cases. The limited accuracy when predicting the *client* class might be explained by the limited number of examples of this class in the dataset.



(a) Avg. client error rate.

(b) Avg. server error rate.

Figure 5.22 – Average `client` and `server` error rate when predicting for a new number of clients and a new setup for different metrics selections and different classifiers for the LAMP use case.

5.3.7 Summary of the comparison of metrics selections for the data processing pipeline and the Web stack

The main goal of this section was to verify whether the main results obtained in Chapter 4 for the data processing pipeline are also valid on the LAMP stack. Besides building an accurate prediction model, we studied the accuracy of the predictions that can be achieved when considering different types of metrics as input for the models. We summarize the results obtained from this point of view in two tables. Table 5.8 shows the subsets of metrics that give the best accuracy for each prediction scenario for the LAMP stack along with the results for the data processing pipeline. Table 5.9 summarizes the metrics types that classifiers select as their main features for predicting performance bottlenecks in both the data processing pipeline and the LAMP stack use cases. The results from our study show that:

- Models based on *recommended* metrics achieve the best accuracy for most classifiers for the LAMP stack. This shows the capacity of combining *reliable* metrics to provide information about the bottlenecks in the system (see Table 5.8).
- Models based on *resource consumption* and *WT* metrics achieve the lower accuracy compared with the baseline and the other models for the data processing pipeline and the LAMP stack use cases. This confirms that combining metrics of different kinds can help in identifying performance bottlenecks in both systems.
- Models that give the best accuracy for both the data processing pipeline and the LAMP stack differ based on the tested case. In general, models based on *recommended* and *server-side-only* metrics give good accuracy with the RF classifier for both systems.
- Models based on specific types of metrics (i.e., *in/out data*) achieve good accuracy for the LAMP stack. The reason behind that might be that metrics belonging to this kind are often reliable indicators of performance bottleneck and combining these metrics can cover more than 60% of the studied setups of the LAMP stack.
- Table 5.9 shows that the *Error Rate (ER)* and the *In/Out Data (IOD)* metric types are useful to analyze the behavior of a multi-tier web stack and to identify its performance bottlenecks. However,

	Data processing pipeline	LAMP stack
Test on a new setup		
Model	Recommended, Server-side-only	Recommended
Test on a new number of clients		
Model	All, Server-side-only	Server-side-only
Test on a new number of clients and a new setup		
Model	Recommended, Server-side-only	All, Recommended

Table 5.8 – Summary of models that give the best accuracy for the data processing pipeline and the LAMP stack.

Metric Type	Data processing pipeline	LAMP stack
Resource Consumption (RC)		
Idle Threads (ITD)	✓	
Error Rate (ER)	✓	✓
Queue Waiting Time (QWT)	✓	
Queue Size (QS)	✓	
Latency (LY)	✓	
Processing Time (PT)		
In/Out Data (IOD)	✓	✓
Uncategorized (UC)		

Table 5.9 – Summary of top 5 metrics types for all use cases of the data processing pipeline and the LAMP stack.

metrics from these different kinds should be combined to get very accurate results. On the other hand, resource consumption metrics are of little help in our studied use cases. These two types of metrics are common for both studied systems, the data processing pipeline, and the LAMP stack.

5.4 Conclusion

In this chapter, we presented a study of a LAMP Web stack comprising two main components: Apache server and MySQL. The main goal of this chapter was to (i) check the robustness of the two proposed approaches when examining a different system in terms of characteristics and workload, and (ii) investigate if it is possible to generalize the findings obtained on the data processing pipeline to other systems.

We applied the analytical approach to the LAMP stack with different setups. The main results are summarized as follows:

- Resource consumption metrics are never *reliable* indicators of performance bottleneck in any of the tested setups of the LAMP stack.
- For each setup of the LAMP stack, there are always some metrics that are *reliable* indicators of performance bottleneck.
- Combining metrics of the **Error Rate (ER)** and the **In/Out Data (IOD)** types is sufficient to cover all the setups of the LAMP stack. This implies that a combination of different metrics types is necessary to identify performance bottleneck in the LAMP stack (see Table 5.6).

- Comparing the types of the reliable metrics obtained by the analytical study for the data processing pipeline and for the LAMP stack, we observe that **Idle Threads (ITD)**, **Error Rate (ER)**, and **In/Out Data (IOD)** are common types for the two systems as Table 5.7 shows.

These main results obtained with the LAMP stack are consistent with the results obtained in Chapter 3 when analyzing the data processing pipeline.

We applied the learning-based approach to the LAMP stack. Comparing learning models based on different selections of metrics, we evaluate the capacity of prediction models to generalize to new setups, to a new number of clients, and even to cases where both the setup and the number of clients are new. Our analysis was run considering 4 different machine learning classifiers (Decision Tree, Random Forest, Support Vector Machines, and Logistic Regression) to avoid drawing conclusions that would be specific to one classifier. The main results can be summarized as follows:

- We manage to achieve good predictions accuracy, up to 88%, with models based on *recommended* and *server-side-only* metrics when predicting on a new setup.

- Best results are obtained when generalizing to a new number of clients (accuracy of 91%). Even in the difficult case where we try to predict on a new number of clients and a new configuration, the best prediction in this case is achieved by models based on *server-side-only*, *in/out data*, and *recommended* (accuracy up to 91%). We note here that the results achieved when predicting on a new setup for a new number of clients is better than when simply predicting on a new setup. However, the difference remains small. The good results obtained when predicting on a new setup and a new number of clients might be explained by the fact that the dataset is unbalanced, i.e., there are much more executions where the server is the bottleneck than where the client is the bottleneck.

- The comparison of the results obtained by considering different subsets of metrics as input for the prediction models show that a combination of metrics of different types guarantees a good prediction of performance bottleneck in the LAMP stack use case.

- Metrics that belong to the following types: *Error Rate* and *In/Out Data* are important but should be combined. On the other hand, models based on *resource consumption* metrics never achieve good accuracy in predicting the bottleneck in our use cases.

- The results obtained from the two approaches for the LAMP stack (Sections 5.2 and 5.3) are consistent. Furthermore, the works in these two sections are complementary as the models based on metrics recommended by the results of Section 5.2 are the ones that achieve most often the best accuracy in the predictions. The good results obtained by *server-side-only* models also confirm the results of Section 5.2 that there are enough reliable indicators on the server side to correctly identify performance bottleneck in the LAMP stack.

Finally, as the main results obtained by applying the analytical study and the learning-based approach to both the data processing pipeline and the LAMP stack are consistent, we conclude that (i) The analytical and the learning-based approach are sufficient in identifying reliable indicators of performance bottleneck in a multi-tier distributed systems (i.e., the LAMP stack and the data processing pipeline); (ii) Combining metrics of different type is necessary to accurately identify performance bottleneck in these systems.

These main results may pave the way towards applying the proposed approaches to other distributed systems with different architectural styles and different workloads to draw more general conclusions about

performance bottleneck indicators in multi-tier distributed systems. These conclusions might lead to an automatic and accurate bottleneck identification tool that covers different multi-tier distributed systems.

Chapter 6

Conclusion and Perspectives

Contents

Distributed systems are complex. They are composed of a variety of software components that provide different functionalities. The complexity of these systems emerges from different factors including the architecture style of the components, the diversity of the communications patterns, and the characteristics of the underlying hardware infrastructures. To achieve scalability, each component can be divided into a number of partitions spread on separate machines for parallel processing. For fault tolerance and high availability, each component or partition of a component typically has a number of replicas. Overall, all these components (and their internal replicas and partitions) have many interactions. In this context, understanding the behavior of a distributed system and how its performance can be improved is a challenging discipline.

In this thesis, we studied a popular architecture for distributed applications: the multi-tier architecture. This architecture is used in different contexts including stream data processing and Web stacks. In such an architecture, it can be difficult for a programmer or a user to know whether a system in a given configuration has reached its maximum capacity or whether it could accommodate more clients. Relying on high-level performance metrics such as the system throughput or the latency is not enough as the maximum performance that can be achieved is impacted by many factors including the underlying hardware, the software configurations, the application logic, and the processed data.

In this regard, in this thesis we study how metrics collected at the software and the hardware levels can help to decide whether a multi-tier system has reached its maximum capacity. A large number of metrics can be exported during the execution of a multi-tier system to try to capture its behavior. However, it is unclear which metrics can provide valuable information about the current state of the system. If such metrics exist, we need to understand whether we can always rely on the same metrics in any configuration of the system, or if different metrics should be considered depending on the case. Finally, it is important to understand whether the analysis can be done automatically or if human expertise is always required to draw conclusions from a set of measurements.

Our study considers as the main use case a data processing pipeline composed of popular software components: Kafka, a distributed publish-subscribe messaging system, Spark Streaming, a Big Data

processing and analysis framework, and Cassandra, a distributed NoSQL database management system. To assess the validity of the results obtained considering the data processing pipeline, the study is then extended to the case of a web stack (i.e., the LAMP stack).

6.1 Contributions

This thesis presents three main contributions.

An analytical study to identify reliable indicators of performance bottlenecks

In this contribution, we studied a large set of metrics exported at the hardware and the software levels considering a large number of configurations of the data processing pipeline. More precisely, we studied three main uses cases of the data processing pipeline including a Kafka cluster, a two-tier system composed of Kafka and Spark Streaming, and a three-tier system composed of Kafka, Spark Streaming, and Cassandra (with different configuration setups of each software). We considered 35 different setups in total. We examined three different workload applications that vary in their characteristics between applying simple operations on the input data (i.e., Wordcount) to using machine learning techniques (i.e., Flight delay prediction). We perform our experiments on top of Grid'5000 [1].

To analyze the evolution of the exported metrics, we proposed a classification of metrics that rates their ability to be useful in evaluating the current state of a multi-tier system. We define four representative categories of patterns based on trend changes in metrics behavior: (i) We define *reliable* metrics as metrics for which the behavior significantly changes at the point where the bottleneck shifts from the client side to the server side; (ii) We define *partially reliable* metrics as metrics for which the trend of the metric evolution needs to be observed over a period of time to allow concluding about the limiting component in the system; (iii) We define *not reliable* metrics as metrics for which the behavior does not change when the multi-tier system reaches its maximum capacity; (iv) We define *misleading* metrics as metrics for which the change in the behavior does not occur at the moment where the bottleneck shifts from the client side to the server side.

The main results obtained through this study show that: (i) Basic resources consumptions metrics are never good indicators of performance bottlenecks in the data processing pipeline; (ii) A combination of metrics of multiple types is required to cover a large number of setups of the data processing pipeline; (iii) Considering a few metrics is, in general, sufficient to ensure that at least one metric will behave reliably in any setup of the data processing pipeline; (iv) Even if taking into account metrics on the client side can provide valuable information about the state of the system, analyzing a few carefully chosen metrics on the server side is enough to conclude in most of the cases of the data processing pipeline.

A learning-based approach for performance bottleneck identification

In the second part of our work, we evaluated the use of machine learning classification algorithms to decide whether a given state of the data processing pipeline is one in which the server has reached its maximum capacity. We considered multiple classification algorithms including Decision Tree, Random Forest, Support Vector Machine, and Logistic Regression. We compared models built using different subsets of metrics as input. Among the metrics selections we evaluated, we considered a case where only metrics recommended by our analytical study are selected. We also considered a case where only server-side software-level metrics are

selected. Finally, we also considered models that only use a single kind of metrics (e.g., resource consumption metrics). We applied the learning-based approach to all the setups considered in the previous analysis of the data processing pipeline.

The obtained results show the ability of a model to generalize to a new setup with a maximum average accuracy of 72% (vs. 54% for a baseline that always predicts that the server has reached its maximum capacity). It also shows the ability of a model to generalize to a new number of clients with a maximum average accuracy of 81% (vs. 54% for the baseline). It finally shows the ability of a model to generalize to a new setup and a new number of clients with a maximum average accuracy of 69% (vs. 54% for the baseline). Taking into account the existence of the gray zone where identifying the server as the bottleneck can be considered as correct, although increasing the number of clients can also improve the throughput, the accuracy of our proposed models reaches even higher scores with error rates below 10% for the difficult cases where we predict on a new setup and a new number of clients.

Moreover, the comparison of the results obtained by models based on different subsets of metrics show that: (i) Models based on *recommended* metrics are the best in terms of prediction accuracy; (ii) Models based only on server metrics manage to achieve good results; (iii) Models based on a single kind of metrics (e.g., resource consumption metrics or waiting time metrics) are the ones that achieve the lowest prediction accuracy in general. As such, the results obtained during this analysis confirm the validity and the usefulness of the analysis presented in Chapter 3.

Towards a general approach of bottleneck identification

In the last contribution, we verify that the results obtained in the previous two contributions are not specific to the data processing pipeline case. Thus, we apply the analytical study and the learning-based approach to a LAMP Web stack comprising Apache server, PHP application, and MySQL.

The results obtained for both approaches confirm the conclusions drawn based on the analysis of the data processing pipeline. More precisely, the analysis for both systems show that: (i) Basic resources consumptions metrics are never reliable indicators of performance bottlenecks; (ii) A combination of metrics of multiple types is required to cover a large number of setups; (iii) Models based on *recommended* metrics achieve the best prediction accuracy for most setups. However, for the LAMP use case, we observed that contrary to the results we obtained on the data processing pipeline, considering models based on specific kind of metrics (namely, *in/out data* metrics) achieve good accuracy (up to 84% vs. 78% for the baseline). The reason behind that might be that metrics belonging to this type are reliable indicators of performance bottleneck and combining these metrics can cover more than 60% of the studied setups of the LAMP stack.

6.2 Perspectives

In light of the results presented in this thesis, we describe four main research directions that would be interesting to follow.

Extend the work to other cases of multi-tier systems

It would be interesting to try to generalize the findings of the two proposed approaches to other stacks, to other software components, or to other applications and communication patterns. For instance, considering the data processing pipeline, in the current applications, Spark workers write the processed

data to the database. Considering applications where spark workers also need to read data from the database component, might lead to different results. This kind of application adds a different kind of communications where a spark worker is requesting the database and waiting for a reply. A potential generalization can also be done by considering some alternatives of one or more components of the data processing pipeline (i.e., replace Kafka by RabbitMQ [13], Spark by Flink [20], or Cassandra by Hbase [16]).

Automatize the analysis of metrics to improve its reliability

The analysis of metrics includes some manual steps including defining the limits of a gray zone where it is unclear whether the server has reached its maximum capacity or not yet. A proper definition of this zone can be important in some cases to decide whether a metric can be considered as reliable or if it is misleading. The parameters that allow defining the limit of this gray zone change depending on the setup and are set manually in our study. Thus, studying how to automatically identify this zone would be interesting to investigate. Several works have been done in this regard. More precisely, authors in [84, 89, 107, 112, 139] present some approaches to detect changes in the throughput in networking and distributed systems domains. Some of these approaches rely on building a statistical model based on changes in the data stream rate. Further research is needed to investigate these works in-depth and check whether we can apply or borrow some principles for our use case.

Build a system that can make more precise recommendations to solve performance issues

The current study focuses on solutions to determine whether the server has reached its maximum capacity in a given configuration of a multi-tier distributed system. Another challenging problem would be to build a tool that suggests some recommendations about which configuration parameter to change to improve the system performance. In this thesis, we studied the factor of provisioning more hardware resources to identify the limiting component in the system. Other factors, such as increasing the number of partitions in the Kafka component would work in some cases. We believe that building a tool that would be able to make such recommendations would be very valuable. A preliminary step for such a tool is to identify the component that is limiting the performance of the system. The results of our analysis tend to show that this should be doable through a few reliable software metrics.

Optimize the placement of distributed systems components

In this thesis, we considered the case where each component of a multi-tier system is deployed on a dedicated machine. However, in practice, for data processing pipelines, it can be the case that multiple components are co-located on the same physical resources. Further research is needed to extend our study to this context by: (i) Studying if our study about reliable indicators of performance bottlenecks is still valid for the placement problem; (ii) Studying whether, by applying a similar method to the one proposed in this thesis, a tool to make recommendations about the placement of components could be built.

Bibliography

[1] Grid'5000. http://www.grid5000.fr. Visited on 2020-01-09.

[2] International business machines corporation. https://ibm.com, 1911. Visited on 2020-01-09.

[3] Akka. https://akka.io/, 1984. Visited on 2020-01-09.

[4] ebay. https://www.ebay.com, 1995. Visited on 2020-01-09.

[5] Mysql. https://https://www.mysql.com/, 1995. Visited on 2020-01-09.

[6] Postgresql. https://www.postgresql.org, 1996. Visited on 2020-01-09.

[7] Netflix, production company. https://www.netflix.com, 1997. Visited on 2020-01-09.

[8] Vmware. https://vmware.com, 1998. Visited on 2020-01-09.

[9] Linkedin. https://www.linkedin.com, 2002. Visited on 2020-01-09.

[10] Amazon kinesis. https://aws.amazon.com/kinesis, 2006. Visited on 2020-01-09.

[11] Spotify. https://www.spotify.com, 2006. Visited on 2020-01-09.

[12] Apache activemq. https://activemq.apache.org, 2007. Visited on 2020-01-09.

[13] Rabbitmq. https://www.rabbitmq.com, 2007. Visited on 2020-01-09.

[14] Airbnb. https://aibnb.com, 2008. Visited on 2020-01-09.

[15] Apache cassandra. https://cassandra.apache.org/, 2008. Visited on 2020-01-09.

[16] Apache hbase. https://hbase.apache.org, 2008. Visited on 2020-01-09.

[17] Airline on time data. http://stat-computing.org/dataexpo/2009/the-data.html, 2009. Visited on 2020-01-09.

[18] Mongodb. https://www.mongodb.com, 2009. Visited on 2020-01-09.

[19] Uber technologies inc. https://uber.com, 2009. Visited on 2020-01-09.

[20] Apache flink. https://flink.apache.org, 2011. Visited on 2020-01-09.

[21] Apache hadoop. https://hadoop.apache.org, 2011. Visited on 2020-01-09.

[22] Apache kafka. https://kafka.apache.org, 2011. Visited on 2020-01-09.

[23] Apache storm. https://storm.apache.org, 2011. Visited on 2020-01-09.

[24] Migrating netflix from datacenter oracle to global cassandra. https://www.slideshare.net/adrianco/migrating-netflix-from-oracle-to-global-cassandra, 2011. Visited on 2020-01-09.

[25] Archive team: The twitter stream grab. https://archive.org/details/twitterstream, 2012. Visited on 2020-01-09.

[26] Cassandra at ebay - cassandra summit 2012. https://www.slideshare.net/jaykumarpatel/cassandra-at-ebay-13920376, 2012. Visited on 2020-01-09.

[27] Apache kafka at spotify. https://www.meetup.com/stockholm-hug/events/121628932, 2013. Visited on 2020-01-09.

[28] Databricks. https://databricks.com, 2013. Visited on 2020-01-09.

[29] Apache spark. https://spark.apache.org, 2014. Visited on 2020-01-09.

[30] Apache spark streaming. https://spark.apache.org/docs/latest/streaming-programming-guide.html, 2014. Visited on 2020-01-09.

[31] What is a monolith? http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html, 2014. Visited on 2020-01-09.

[32] Apache kafka. https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster, 2015. Visited on 2020-01-09.

[33] How to use mytop to monitor mysql performance. https://www.digitalocean.com/community/tutorials/how-to-use-mytop-to-monitor-mysql-performance, 2015. Visited on 2020-02-05.

[34] Improvements to kafka integration of spark streaming. https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html, 2015. Visited on 2020-01-09.

[35] Airstream: Spark streaming at airbnb- spark summit 2016. https://databricks.com/session/airstream-spark-streaming-at-airbnb, 2016. Visited on 2020-01-09.

[36] Apache kafka powered by. https://cwiki.apache.org/confluence/display/KAFKA/Powered+By, 2016. Visited on 2020-01-09.

[37] Monitoring mysql performance metrics. https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics, 2016. Visited on 2020-01-09.

[38] Apache kafka compression. https://cwiki.apache.org/confluence/display/KAFKA/Compression, 2017. Visited on 2020-01-09.

[39] Cassandra architecture and write path anatomy. https://medium.com/jorgeacetozi/cassandra-architecture-and-write-path-anatomy-51e339bcfe0c, 2017. Visited on 2020-01-09.

[40] Datastax spark cassandra connector. https://github.com/datastax/spark-cassandra-connector, 2017. Visited on 2020-01-09.

[41] Kafka concepts and common patterns. https://www.beyondthelines.net/computing/kafka-patterns/, 2017. Visited on 2020-01-09.

[42] Monitoring apache web server performance. https://www.datadoghq.com/blog/monitoring-apache-web-server-performance, 2017. Visited on 2020-01-09.

[43] Multi-processing modules in apache. https://www.datadoghq.com/blog/collect-apache-performance-metrics/, 2017. Visited on 2020-01-09.

[44] Spark performance optimization: shuffle tuning. http://bigdatatn.blogspot.com/2017/05/spark-performance-optimization-shuffle.html, 2017. Visited on 2020-01-09.

[45] Airstream: Spark streaming at airbnb. https://www.slideshare.net/JenAman/airstream-spark-streaming-at-airbnb, 2018. Visited on 2020-01-09.

[46] Near real-time netflix recommendations using apache spark streaming. https://databricks.com/session/near-real-time-netflix-recommendations-using-apache-spark-streaming, 2018. Visited on 2020-01-09.

[47] Project and product names using spark. http://spark.apache.org/powered-by.html, 2018. Visited on 2020-01-09.

[48] Spark streaming vs flink vs storm vs kafka streams vs samza : Choose your stream processing framework. https://www.linkedin.com/pulse/spark-streaming-vs-flink-storm-kafka-streams-samza-choose-prakash/, 2018. Visited on 2020-01-09.

[49] Using spark streaming and nifi for the next generation of etl in the enterprise. https://www.youtube.com/watch?v=WwjmUAszB-0, 2018. Visited on 2020-01-09.

[50] Alternatives to apache cassandra. https://alternativeto.net/software/cassandra-database/, 2019. Visited on 2020-01-09.

[51] Alternatives to apache kafka. https://stackshare.io/kafka/alternatives, 2019. Visited on 2020-01-09.

[52] Apache zookeeper. https://zookeeper.apache.org, 2019. Visited on 2020-01-09.

[53] Building the new twitter.com. https://blog.twitter.com/engineering/en_us.html, 2019. Visited on 2020-01-09.

[54] Java management extensions. https://en.wikipedia.org/wiki/JMX, 2019. Visited on 2020-01-09.

[55] Lamp stack architecture. https://www.liquidweb.com/kb/what-is-a-lamp-stack/, 2019. Visited on 2020-01-09.

[56] Perf. http://www.brendangregg.com/perf.html, 2019. Visited on 2020-01-09.

[57] Random forest. https://towardsdatascience.com/understanding-random-forest-58381e0602d2, 2019. Visited on 2020-02-05.

[58] Understanding logistic regression. https://www.datacamp.com/community/tutorials/understanding-logistic-regression-python, 2019. Visited on 2020-02-05.

[59] Apache server mod. https://httpd.apache.org/docs/2.4/mod/worker.html, 2020. Visited on 2020-01-09.

[60] Apache web server. https://stack.g2.com/apache-web-server, 2020. Visited on 2020-01-09.

[61] Apache web server features. https://en.wikipedia.org/wiki/Comparison_of_web_server_software#Features, 2020. Visited on 2020-01-09.

[62] Innodb multiple buffer pools. https://dev.mysql.com/doc/refman/5.5/en/innodb-multiple-buffer-pools.html, 2020. Visited on 2020-01-09.

[63] An introduction to implicit invocation architectures. https://www.cfconf.org/fusebox2003/talks/mach-iib.pdf, 2020. Visited on 2020-01-09.

[64] Mysql. https://stackshare.io/mysql, 2020. Visited on 2020-01-09.

[65] Charu C Aggarwal. *Data classification: algorithms and applications*. CRC press, 2014.

[66] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014.

[67] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *ACM Sigplan Notices*, volume 45, pages 739–753. ACM, 2010.

[68] Sylvain Arlot, Alain Celisse, et al. A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79, 2010.

[69] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM international conference on management of data (SIGMOD)*, pages 1383–1394. ACM, 2015.

[70] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 10, pages 1–14, 2010.

[71] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. Sentiwordnet 3.0: an enhanced lexical resource for sentiment analysis and opinion mining. In *International Conference on Language Resources and Evaluation (LREC)*, volume 10, pages 2200–2204, 2010.

[72] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[73] Muli Ben-Yehuda, David Breitgand, Michael Factor, Hillel Kolodner, Valentin Kravtsov, and Dan Pelleg. Nap: a building block for remediating performance bottlenecks via black box network analysis. In *Proceedings of the 6th international conference on Autonomic computing (ICAC)*, pages 179–188. ACM, 2009.

[74] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. Debugging distributed systems. *Queue*, 14(2):50, 2016.

[75] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems.* " O'Reilly Media, Inc.", 2016.

[76] Betsy Beyer, Niall Richard Murphy, David K Rensin, Kent Kawahara, and Stephen Thorne. *The Site Reliability Workbook: Practical Ways to Implement SRE.* " O'Reilly Media, Inc.", 2018.

[77] Christopher M Bishop. *Pattern recognition and machine learning.* Springer Science, Business Media, 2006.

[78] Peter Bodík, Rean Griffith, Charles A Sutton, Armando Fox, Michael I Jordan, and David A Patterson. Statistical machine learning makes automatic control practical for internet datacenters. *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 9:12–12, 2009.

[79] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[80] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. Classification and regression trees. 1984.

[81] Marcio Castro, Luis Fabricio Wanderley Goes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean-Francois Mehaut. A machine learning-based approach for thread mapping on transactional memory applications. In *18th International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2011.

[82] Alain Celisse et al. Optimal cross-validation in density estimation with the l^2-loss. *The Annals of Statistics*, 42(5):1879–1910, 2014.

[83] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[84] Loon-Been Chen and I-Chen Wu. Detection of summative global predicates. *IEICE Transactions on Information and Systems*, 86(5):976–980, 2003.

[85] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *Proceedings of International Conference on Autonomic Computing (ICAC)*, pages 36–43. IEEE, 2004.

[86] I-Hsin Chung, Guojing Cong, David Klepacki, Simone Sbaraglia, Seetharami Seelam, and Hui-Fang Wen. A framework for automated performance bottleneck detection. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–7. IEEE, 2008.

[87] Ira Cohen, Jeffrey S Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 4, pages 16–16, 2004.

[88] OW2 Consortium et al. Rubis: Rice university bidding system. *URL http://rubis. ow2. org*, 2013.

[89] Graham Cormode and Shanmugavelayutham Muthukrishnan. What's new: Finding significant differences in network data streams. *IEEE/ACM Transactions on Networking*, 13(6):1219–1232, 2005.

[90] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[91] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[92] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[93] Nicolas Denoyelle, Brice Goglin, Emmanuel Jeannot, and Thomas Ropars. Data and thread placement in numa architectures: A statistical learning approach. In *International Conference on Parallel Processing (ICPP)*. ACM, 2019.

[94] IBM Cloud Education. Lamp stack. https://www.ibm.com/cloud/learn/lamp-stack-explained, 2019. Visited on 2020-01-09.

[95] Byron Ellis. *Real-time analytics: Techniques to analyze and visualize streaming data*. John Wiley & Sons, 2014.

[96] Raul Estrada and Isaac Ruiz. Big data smack. *Apress, Berkeley, CA*, 2016.

[97] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[98] Roy T. Fielding and Gail Kaiser. The apache http server project. *IEEE Internet Computing*, 1(4):88–90, 1997.

[99] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[100] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 19–33. ACM, 2019.

[101] C Lee Giles, Steve Lawrence, and Ah Chung Tsoi. Noisy time series prediction using recurrent neural networks and grammatical inference. *Machine learning*, 44(1-2):161–183, 2001.

[102] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[103] Eben Hewitt. *Cassandra: the definitive guide*. " O'Reilly Media, Inc.", 2010.

[104] Steve Hoffman. *Apache Flume: distributed log collection for Hadoop*. Packt Publishing Ltd, 2013.

[105] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[106] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[107] Dino Ienco, Albert Bifet, Bernhard Pfahringer, and Pascal Poncelet. Change detection in categorical evolving data streams. In *Proceedings of the 29th annual ACM symposium on applied computing (SAC)*, pages 792–797, 2014.

[108] Waheed Iqbal, Matthew N Dailey, David Carrera, and Paul Janecek. Sla-driven automatic bottleneck detection and resolution for read intensive multi-tier applications hosted on a cloud. In *International Conference on Grid and Pervasive Computing (GPC)*, pages 37–46. Springer, 2010.

[109] Gueyoung Jung, Galen Swint, Jason Parekh, Calton Pu, and Akhil Sahai. Detecting bottleneck in n-tier it applications through analysis. In *International Workshop on Distributed Systems: Operations and Management (DSOM)*, pages 149–160. Springer, 2006.

[110] Dharmesh Kakadia. *Apache Mesos Essentials*. Packt Publishing Ltd, 2015.

[111] A Karve, Tracy Kimbrel, Giovanni Pacifici, Mike Spreitzer, Malgorzata Steinder, Maxim Sviridenko, and A Tantawi. Dynamic placement for clustered web applications. In *Proceedings of the 15th international conference on World Wide Web (WWW)*, pages 595–604. ACM, 2006.

[112] Gunjan Khanna, Ignacio Laguna, Fahad A Arshad, and Saurabh Bagchi. Stateful detection in high throughput distributed systems. In *26th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 275–287. IEEE, 2007.

[113] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[114] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the6th Workshop on Networking meetsDatabases (NetDB)*, pages 1–7, 2011.

[115] Christoph Lameter et al. Numa (non-uniform memory access): An overview. *Acm queue*, 11(7):40, 2013.

[116] Donghun Lee, Sang K Cha, and Arthur H Lee. A performance anomaly detection and analysis framework for dbms development. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(8):1345–1360, 2011.

[117] Donghun Lee, Sang K Cha, and Arthur H Lee. A performance anomaly detection and analysis framework for dbms development. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(8):1345–1360, 2012.

[118] David Linthicum. Chapter 1: Service oriented architecture (soa). *SOAs are like snowflakes–no two are alike.","" https://msdn. microsoft. com/en-us/library/bb833022. aspx*, 2016.

[119] Wei-Yin Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.

[120] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 378–393. ACM, 2015.

[121] Simon Malkowski, Markus Hedwig, Jason Parekh, Calton Pu, and Akhil Sahai. Bottleneck detection using statistical intervention analysis. In *International Workshop on Distributed Systems: Operations and Management (DSOM)*, pages 122–134. Springer, 2007.

[122] Simon Malkowski, Markus Hedwig, and Calton Pu. Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 118–127. IEEE, 2009.

[123] Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, et al. *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale*. " O'Reilly Media, Inc.", 2012.

[124] Mohri Mehryar, Rostamizadeh Afshin, and Talwalkar Ameet. Foundations of machine learning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 17, 2012.

[125] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[126] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Burstiness in multi-tier applications: Symptoms, causes, and new models. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 265–286. Springer, 2008.

[127] Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: the definitive guide: real-time data and stream processing at scale*. " O'Reilly Media, Inc.", 2017.

[128] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 184–200. ACM, 2017.

[129] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and V ICSI. Making sense of performance in data analytics frameworks. In *Conference on Symposium on Networked Systems Design & Implementation (NSDI)*, volume 15, pages 293–307, 2015.

[130] Wei Pan, Zhanhuai Li, Yansong Zhang, and Chuliang Weng. The new hardware development trend and the challenges in data management and analysis. *Data Science and Engineering (DSE)*, 3(3):263–276, 2018.

[131] Jason Parekh, Gueyoung Jung, Galen Swint, Calton Pu, and Akhil Sahai. Issues in bottleneck detection in multi-tier enterprise applications. In *14th IEEE International Workshop on Quality of Service (IWQoS)*, pages 302–303. IEEE, 2006.

[132] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[133] Rob Powers, Moises Goldszmidt, and Ira Cohen. Short term performance forecasting in enterprise systems. In *Proceedings of the eleventh ACM international conference on Knowledge discovery in data mining (SIGKDD)*, pages 801–807. ACM, 2005.

[134] Jia Rao and Cheng-Zhong Xu. Online measurement of the capacity of multi-tier websites using hardware performance counters. In *The 28th International Conference on Distributed Computing Systems (ICDCS)*, pages 705–712. IEEE, 2008.

[135] Mikael Ronstrom and Lars Thalmann. Mysql cluster architecture overview. *MySQL Technical White Paper*, 2004.

[136] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[137] Raja R Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC)*, pages 401–414. ACM, 2016.

[138] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High performance MySQL: optimization, backups, and replication*. " O'Reilly Media, Inc.", 2012.

[139] Robert Schweller, Yan Chen, Elliot Parsons, Ashish Gupta, Gokhan Memik, and Yin Zhang. Reverse hashing for sketch-based change detection on high-speed networks. Technical report, 2004.

[140] Chris Scollo and Sascha Shumann. *Professional PHP programming*. Wrox Press Ltd., 1999.

[141] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[142] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[143] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH computer architecture news*, volume 23, pages 392–403. ACM, 1995.

[144] Tien Van Do, Udo R Krieger, and Ram Chakka. Performance modeling of an apache web server with a dynamic pool of service processes. *Telecommunication Systems*, 39(2):117–129, 2008.

[145] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC)*, page 5. ACM, 2013.

[146] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *IEEE Symposium on Security and Privacy (SP)*, pages 36–52. IEEE, 2018.

[147] Chengwel Wang, Soila P Kavulya, Jiaqi Tan, Liting Hu, Mahendra Kutare, Mike Kasick, Karsten Schwan, Priya Narasimhan, and Rajeev Gandhi. Performance troubleshooting in data centers: an annotated bibliography. *ACM SIGOPS Operating Systems Review*, 47(3):50–62, 2013.

[148] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 31–40. IEEE, 2013.

[149] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. An experimental study of rapidly alternating bottlenecks in n-tier applications. In *IEEE Sixth International Conference on Cloud Computing (CLOUD)*, pages 171–178. IEEE, 2013.

[150] Matt Welsh and David E Culler. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 4–4. Seattle, WA, 2003.

[151] Pengcheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. vperfguard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 271–282. ACM, 2013.

[152] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 619–634, 2016.

[153] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 244–259. ACM, 2013.

[154] Wei Xu, Ling Huang, and Michael I Jordan. Experience mining google's production console logs. In *Proceedings of SLAML*, 2010.

[155] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–172. ACM, 2011.

[156] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles (SOSP)*, pages 423–438. ACM, 2013.

[157] Cha Zhang and Yunqian Ma. *Ensemble machine learning: methods and applications*. Springer, 2012.

[158] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 415–432. ACM, 2019.

[159] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. *arXiv preprint arXiv:1903.06631*, 2019.

[160] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 379–391. ACM, 2013.

[161] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, pages 149–161. ACM, 2018.

[162] Tao Zou, Ronan Le Bras, Marcos Vaz Salles, Alan Demers, and Johannes Gehrke. Cloudia: a deployment advisor for public clouds. *The VLDB Journal*, 24(5):633–653, 2015.