Optimization of software license placement in the Cloud for economical and efficient deployment

Arthur Chevalier

Numéro National de Thèse : 2020LYSEN071

THESE de DOCTORAT DE L'UNIVERSITE DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale *N°*512

École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 24/11/2020, par :

Arthur CHEVALIER

Optimisation du placement des licences logicielles dans le Cloud pour un déploiement économique et efficient

Devant le jury composé de :

Frédéric DESPREZ	Directeur de recherche	Inria	*Rapporteur*
Adrien LEBRE	Professeur des universités	IMT Atlantique	*Rapporteur*
Arnaud LEGRAND	Directeur de recherche	LIG	*Rapporteur*
Fabienne BOYER	Maître de conférences HDR	LIG	*Examinatrice*
Eddy CARON	Maître de conférences HDR	LIP	*Directeur*
Noëlle BAILLON-BACHOC	Ingénieure	Orange S.A.	*Co-encadrante*

Contents

Remerciements

Je voudrais commencer par remercier mes directeurs de thèse, Noëlle Baillon et Eddy Caron, qui ont toujours été source de nombreux conseils et qui ont su me soutenir (et me supporter) durant ces trois années de recherche. Je suis très reconnaissant de toute leur expérience qu'ils m'ont apportée au fil des années.

Je voudrais ensuite remercier ma famille qui m'a apporté leur soutien et qui a su me ramener sur terre quand c'était nécessaire. Je voudrais tout particulièrement remercier mes parents qui ont du faire face à de terribles nouvelles, mais qui ont tout de même trouvé le temps de m'encourager.

Ensuite, je ne peux que remercier mes amis, pour leurs attentions et encouragements ainsi qu'à toutes ses soirées mémorables. Je remercie chaleureusement aussi toutes les personnes de l'Inria Bordeaux qui m'ont donné le goût de la recherche et qui m'ont permis de faire cette thèse.

Par ailleurs, je voudrais remercier mon équipe chez Orange et tout particulièrement Anne-Lucie et Julien pour toutes les joyeuses discussions, les bons moments et surtout le mercre-dé qui a su guider parfaitement nos journées.

De plus, je veux remercier toutes les personnes que j'ai rencontré durant ces années de recherche, aux écoles d'hiver, dans la super équipe des students volunteer, au DIU, et tant d'autres, pour tous les bons moments passés et les expériences dont je me rappellerai longtemps.

Je voudrais également grandement remercier Frédéric Desprez, Adrien Lebre et Arnaud Legrand, pour l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de ma thèse ainsi que mon jury pour le temps qu'ils ont accepté de m'accorder en cette période difficile.

Enfin, je ne saurai pas te remercier assez, Annah, pour ta patience, ton soutien quotidien inestimable, tes attentions et encouragements dont tu as fait preuve durant ces années.

Résumé

Cette thèse s'intéresse au Software Asset Management (SAM) qui correspond à la gestion de licences, de droits d'usage, et du bon respect des règles contractuelles. Lorsque l'on parle de logiciels propriétaires, ces règles sont bien souvent mal interprétées ou totalement incomprises. En échange du fait que nous sommes libres de licencier notre usage comme bon nous semble, dans le respect du contrat, les éditeurs possèdent le droit d'audit. Ces derniers peuvent vérifier le bon respect des règles et imposer, lorsque ces dernières ne sont pas respectées, des pénalités bien souvent d'ordre financières. Cela peut mener à des situations désastreuses comme le procès entre AbInBev et SAP où ce dernier réclamait 600 millions de dollars de pénalité. L'émergence du Cloud a grandement augmenté la problématique du fait que les droits d'usages des logiciels n'étaient pas initialement prévus pour ce type d'architecture. Après un historique académique et industriel du Software Asset Management, des racines aux travaux les plus récents concernant le Cloud et l'identification logicielle, nous nous intéressons aux méthodes de licensing des principaux éditeurs comme Oracle, IBM et SAP avant d'introduire les différents problèmes intrinsèques au SAM. Le manque de standardisation dans les métriques, des droits d'usages spécifiques, et la différence de paradigme apportée par le Cloud et prochainement le réseau virtualisé rendent la situation plus compliquée qu'elle ne l'était déjà. Nos recherches s'orientent vers la modélisation de ces licences et métriques afin de s'abstraire du côté juridique et flou des contrats. Cette abstraction nous permet de développer des algorithmes de placement de logiciels qui assurent le bon respect des règles contractuelles en tout temps. Ce modèle de licence nous permet également d'introduire une heuristique de déploiement qui optimise plusieurs critères au moment du placement du logiciel tels que la performance, l'énergie et le coût des licences. Nous introduisons ensuite les problèmes liés au déploiement de plusieurs logiciels simultanément en optimisant ces mêmes critères et nous apportons une preuve de la NP-complétude du problème de décision associé. Afin de répondre à ces critères, nous présentons un algorithme de placement qui approche l'optimal et utilise l'heuristique ci-dessus. En parallèle, nous avons développé un logiciel SAM qui utilise ces recherches pour offrir une gestion automatisée et totalement générique des logiciels dans une architecture Cloud. Tous ces travaux ont été menés en collaboration avec Orange et testés lors de différentes preuves de concept avant d'être intégrés totalement dans l'outillage SAM.

Abstract

This thesis takes place in the field of Software Asset Management, license management, use rights, and compliance with contractual rules. When talking about proprietary software, these rules are often misinterpreted or totally misunderstood. In exchange for the fact that we are free to license our use as we see fit, in compliance with the contract, the publishers have the right to make audits. They can check that the rules are being followed and, if they are not respected, they can impose penalties, often financial penalties. This can lead to disastrous situations such as the lawsuit between AbInBev and SAP, where the latter claimed a US$600 million penalty. The emergence of the Cloud has greatly increased the problem because software usage rights were not originally intended for this type of architecture. After an academic and industrial history of Software Asset Management (SAM), from its roots to the most recent work on the Cloud and software identification, we look at the licensing methods of major publishers such as Oracle, IBM and SAP before introducing the various problems inherent in SAM. The lack of standardization in metrics, specific usage rights, and the difference in paradigm brought about by the Cloud and soon the virtualized network make the situation more complicated than it already was. Our research is oriented towards modeling these licenses and metrics in order to abstract from the legal and blurry side of contracts. This abstraction allows us to develop software placement algorithms that ensure that contractual rules are respected at all times. This licensing model also allows us to introduce a deployment heuristic that optimizes several criteria at the time of software placement such as performance, energy and cost of licenses. We then introduce the problems associated with deploying multiple software at the same time by optimizing these same criteria and prove the NP-completeness of the associated decision problem. In order to meet these criteria, we present a placement algorithm that approaches the optimal and uses the above heuristic. In parallel, we have developed a SAM tool that uses these researches to offer an automated and totally generic software management in a Cloud architecture. All this work has been conducted in collaboration with Orange and tested in different Proof-Of-Concept before being fully integrated into the SAM tool.

Chapter 1

Introduction

Contents

Since 1976, software has been subject to intellectual property law. Its editor, whether an individual or a legal entity, is the sole owner. It grants licenses for use rights that are strictly described in general terms and conditions or in specific commercial contracts. The non-respect of these use rights is a crime: counterfeiting. Up to 1980s, few if any licensing strategies were used. Software was simply sold, and editors hoped that it wouldn't be copied. It wasn't such an issue as software was shipped on proprietary format like cassette or tapes and, therefore, were difficult to copy for the mass. The rise of floppy disk countered this situation, and several techniques were developed to answer this problem. Early protection was hard-coded key in the medium, but it didn't prevent multiple users to share it. Then, some techniques bounded the software to the system it ran on by using unique identifier (this technique is still in use today for some cases). With advances in networking, editors shifted to license server organization where the licenses were not bound to machines anymore but controlled and distributed by a networked server. This allowed to propose multiple types of licenses like machine bounded or named licenses where a license was bound to a user. Companies could purchase fewer licenses by selecting the best type that fit their needs. With digital communications, the license keys could be exchanged electronically, as well as illegal copies. Then came the online activation where the software made contact directly with the editor's servers to check if a license was available. In one hand, it allowed to be better at countering piracy but in the other hand editors had sole control over the use of software we contracted. We arrive in a situation where licenses will become no longer physical or bound to keys or files. They are becoming purely virtual and the only thing that will matter is that we need to buy enough licenses to cover our use of the software as we will explain in more details below.

With the digital boom in today's world, the speed at which developments are oc-

curring is becoming too important for companies to keep up with effectively. These companies need to digitalize themselves to stay competitive, and the management of their digital footprint and software assets must be rigorous and perfect. Those that cannot keep up stay on the sidelines and could disappear to the benefit of large companies capable of making such transitions. The software market is worth more than US$650 billion a year and has seen a steady growth of 6% over the past five years. All areas of IT are now sold as Software: applications, security, storage, network infrastructure, and much more meaning that poor management of software is a real problem.

Therefore, to improve the efficiency of its process, its consistency, and its quality, an enterprise must keep track of its software assets. To do it, some tools exist and a trend grows more and more within companies: Software Asset Management.

1.1 Software Asset Management: SAM

According to the Information Technology Infrastructure Library (ITIL), SAM is defined as:

> "... all the infrastructure and processes necessary for the effective management, control and protection of the software assets ... throughout all stages of their life cycle."
>
> (ITIL's Guide to Software Asset Management)

For some small companies or personal use, it will mean a simple monitoring of software, following the installation on employees computers, managing the upgrade to the latest version and keeping the license keys in a secure place. But it becomes highly complex as soon as a business grows a little or if it uses more distributed infrastructure. The more it grows, the more it needs to have an active Software Asset Management following the whole life cycle of the product, starting from the purchasing process, and going all the way through the delivery, the deployment, the updates and upgrades, the migration, the usage and the decommissioning (Figure 1.1).

The life cycle is split in multiple steps in which the software asset manager has to track different information. During the procurement or the purchasing process, we have to get information about contractual rights concerning the product called Product Use Rights (PURs). They are rules to follow when we use the product and are generally globally defined with additional rules when needed. Some examples of PURs can be the language of the product, the region where we can install it, the rights to upgrade (or downgrade), the available options, etc.

As companies get the right to manage their licence stock as they see fit. In return, publishers may conduct audits. It means, that they trust us to follow the rules we contractualized but in return, at any time, they can come and check if it is the case. If they find wrongdoing, they will impose a penalty which is, most of the time, an economical fine. Such penalty can be huge as seen in two recent cases: SAP vs Diageo [1] and SAP vs AbInBev [2]. The penalties applied in these cases were respectively US$65 millions

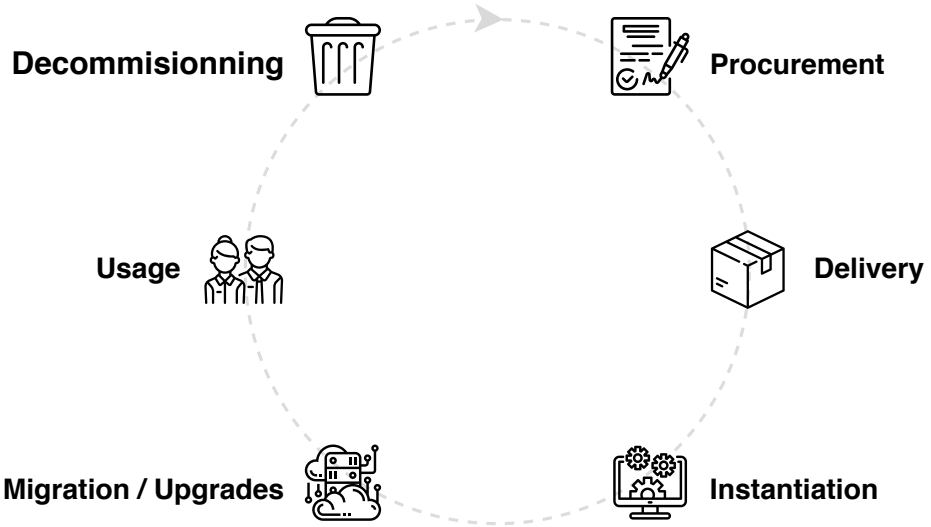


Figure 1.1: Life cycle of a product from buying process to decommissioning.

and US$600 millions, which are colossal amounts for not respecting contractual rights. Another impact of getting a penalty because of noncompliance is that the company will suffer in its public image.

Next, during the delivery of the product we have to effectively keep track of what has been delivered and which contract it is linked to. Most of the time, the delivery is just an executable file (the installer) so it is important to keep trace of it for the following step: the instantiation of the product. During this step, it is vital to link the product installed to our use rights to avoid noncompliant installations as previously seen (Figure 1.2). No physical link exists between the executable and the contract, so the software asset manager must keep a record of which executable corresponds to which contract. Some work have been done to automatize this process and several methods were proposed as we will discuss in Section 1.3.

During the instantiation step, either it's an installation on an employee station or a deployment in the Cloud, it is important to have the details on which product is installed where and getting the details of the platform this product is installed on. To check the compliance with the PURs, we will need a variety of information coming from distinct places like the language of the platform we install on, the options activated or not, the country where the platform is located and so on. All those data are critical because they ensure that compliance testing can be carried out. Theoretically, the software asset manager should oversee instantiation of any product but, in reality, it is not workable because of the amount of data to check and the number of instantiation to follow.

Same thing happen during any migration, upgrade and use of a product. The software asset manager have to check every PUR in the contract to see if the company possess the right to make such a change or use the product. Finally, during the decommissioning of the product, the software asset manager have to check impacts it can make on other

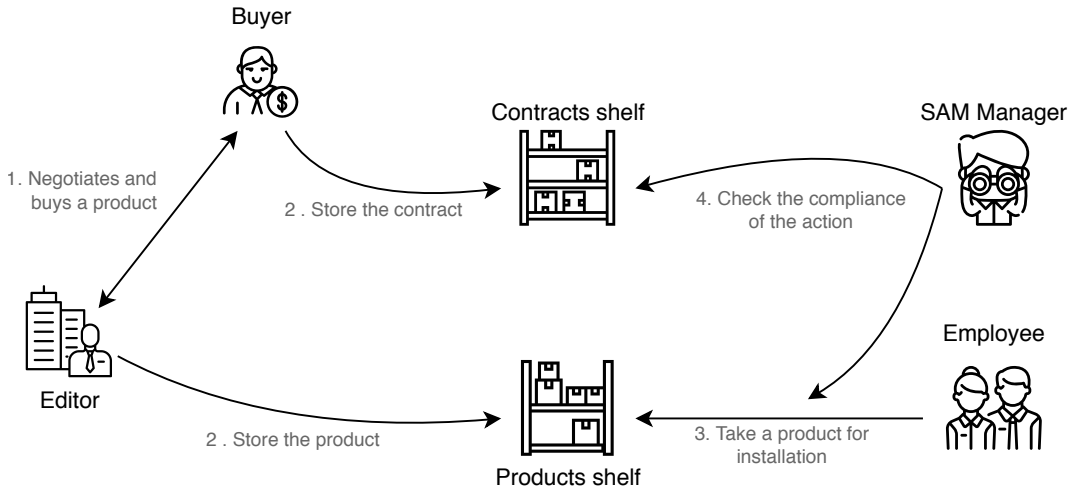


Figure 1.2: Process of buying a product before installating it with the assurance from the software asset manager that it will be compliant.

products as well as if the product was correctly decommissioned. The product 'Acrobat Reader' from Adobe is a good example: We can use an evaluation version to test its functionalities for free. When the evaluation period expires, if we just let the application installed but without activating it, even if we don't use it anymore, it could be considered as a noncompliant installation of the product. Therefore, the decommissioning step has been failed, even if it is a free product.

The most important notion we will focus on during this entire document is licensing. Besides other PURs that describe in which condition we can use our product, we also have to possess licenses for it. These licenses are obtained through contracts and are purely virtual. The important rule is:

Compliance rule: "**We have to possess enough licenses to cover all our products when we use them and respect every PURs**"

It is important to distinguish between a license key (or a license file) and a license. The first two are means for the application to make sure we have the corresponding licenses to be used and if not, will offer little or no functionality. What counts is that we have enough licenses and respect all the PURs described in our contracts. For example, getting rid of the Microsoft Windows watermark ("Activate Windows") by using a valid product key does NOT mean that we are compliant and that our product is respecting the Software License Agreement (SLA). Even if Windows itself says that the product is activated, if we use it only for remote use, we are not respecting one PUR in the terms and services [3]. Excepting when it is specified in the terms and services that we have

to activate the product, we can just use it without activation and make sure that the sum of licenses we bought is enough to cover our use.

Most people think that there is a one-one relation between a software and its license, which is wrong. Except Open-Source Software (OSS) with their specific licenses, we will need, especially for proprietary software, more than one license to have the right to use it meaning that for one user and one product we will have to possess multiple licenses (but maybe one license key to activate the software). To get the number of licenses we need to possess, there is one important and always present PUR: the *metric*. The *metric* is a description of how to compute the number of licenses. It can say that we need 5 licenses per user or 10 licenses per hour of use, for example. In addition, there may be several *metrics* for a single product, for example, the Oracle Database can be licensed with two different *metrics* (Processor *metric* and Named User Plus *metric*). For each product, we have to choose which available *metric* for this product we want to use as they are distinct ways of computation. We need to choose between buying x licenses with the *metric* processor or y licenses with the *metric* NUP. Each *metric* will give a price per license too, so even if $x = y$ the final prices, P_x and P_y can be different. On one hand, it gives the choice to suit the needs. In the other hand, we will have to know every *metric* for each product we use, and compare the prices depending on our use of such product. To better understand this notion, Figure 1.3 is an example with the following definitions of two *metrics* of the Oracle Database coming from the license terms and definitions[1] of Oracle [4]:

Processor The number of required licenses shall be determined by multiplying the total number of cores of the processor by a core processor [5]. All cores on all multicore chips for each licensed program are to be aggregated before multiplying by the appropriate core processor licensing factor and all fractions of a number are to be rounded up to the next whole number.

Named User Plus Requires a minimum of 25 Named User Plus per Processor licenses or the total number of actual users, whichever is greater. When licensing the Oracle Database by Named User Plus, all users who are using the Oracle Database, as well as all non-human operated devices that are accessing the Oracle Database must be licensed. The following licensing rules apply:

- If non-human operated devices such as sensors are connecting to the Oracle Database, then all devices need to be licensed.
- If human-operated devices such as bar code scanners are connecting to the Oracle Database, then all humans operating these devices need to be licensed.

[1] These definitions are for Enterprise Edition and stripped for better clarity.

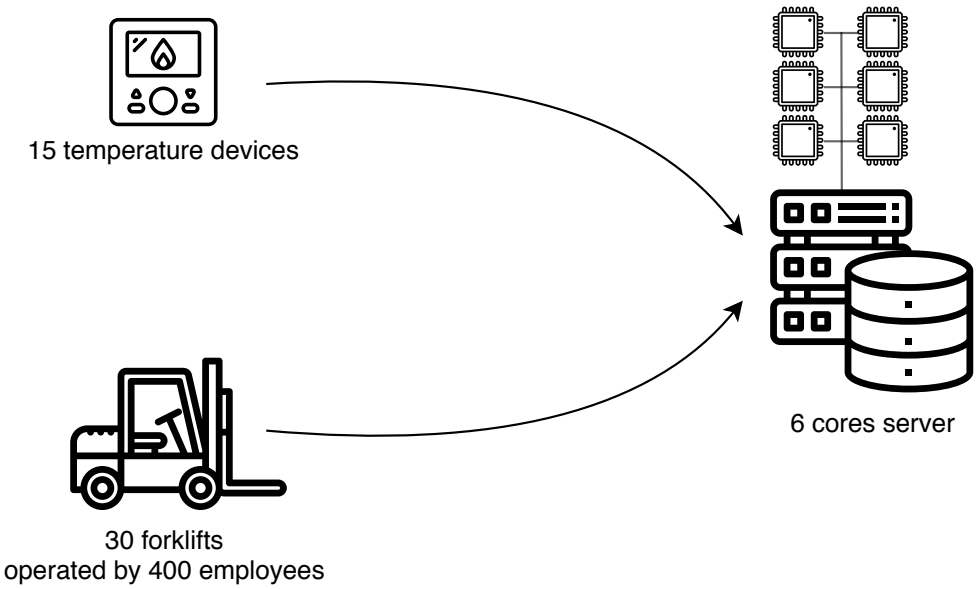


Figure 1.3: Here a database is used to manage data coming from 15 temperature devices and 400 employees manipulating 30 forklifts. As the database is hosted on a 6-cores server, and assuming the corefactor is 1, we would need 6 'processor' licenses. The Named User Plus *metric* requires the maximum between 25 times the number of processor licenses or the number of users. With the licensing rules above, we know that the users are the temperature devices and the employees so the number of 'NUP' licenses is $max(25 \times 6, 400 + 15) = 415$. Therefore, we have to choose between buying 6 processor licenses or 415 NUP licenses. With the public prices [6] of US$47,500 per processor license and US$950 per NUP licenses, it is cheaper to choose the processor license for a total of US$285,000.

- If non-human operated devices and human-operated devices are connecting to the Oracle Database and are mutually exclusive, then all non-human devices and all humans operating devices need to be licensed.

Several problems appear: First, the *metrics* are defined in the contract and as a result are written in contractual language that can be blurry or lead to misunderstandings. For example, a SAP publication from April 2018 [7] defines "use" of its software as "to activate the processing capabilities of the Software, load, execute, access, employ the Software, or display information resulting from such capabilities. Use may occur by way of an interface delivered with or as a part of the Software, a Licensee or third-party interface, or another intermediary system". This definition could potentially say that we are using the SAP product when we make a presentation with one result in a slide computed with this SAP product. The dangerousness of such definition is clear to understand. This definition is SAP's famous indirect-access computation leading to the

two previously announced trials ([1] and [2]) where both companies did not understand correctly and paid full price.

Second, as there is no standard for *metrics*, editors can invent any kind of formula. As a result, the *metrics* are varied and there is potentially hundred per editors, specially for the main ones. While this number grows constantly, the type of data needed for these definitions become more and more diversified. They may relate to infrastructure data (processors, bandwidth capacity, sockets, etc.), companies' information (turnover, number of employees), or even more uncanny data like the number of lines on generated bills or the number of lines resulting a SQL query. This variety of needed information requires the software asset managers to gather and cross-reference vast amount of data.

In addition, with the rise of the Bring Your Own Device (BYOD) practice where employees use their personal devices for professional use, the boundary become blurred for software asset managers between software used in a personal or professional context. As some editors impose specific versions for professional use, a personal use of a proprietary software can lead to noncompliant result of audit becoming a nightmare for SAM processes. Better practices, like 'Corporate-Owned, Personally Enabled' (COPE) where the company buy and provide to employees the devices, are safer because companies can enforce strict security policies unlike BYOD seen previously. Being made possible by these practices, an even good willing employee can install a software breaking some rights or even a pirated one not knowing the impact it can have on its company. We can detect such installation with enough tools and processes, but where do we draw the line between Software Asset Management for company security and a company spying on its employees?

Finally, all the process of checking the rights, computing the number of licenses, getting the right prices, collecting inventories of infrastructure information and software installation is extremely error prone when not automatized. The problem of automatization is that every input of data have to be checked and corrected because a single error of computation can lead to disastrous situations. This check and correction is actually mostly made by humans and, by definition, humans are error prone.

All these considerations need to be addressed before going to scale because, as we will see in the next section, the Cloud further complicates the matter.

1.2 Software Asset Management in the Cloud

Software Asset Management is already not a simple task in traditional architectures. As we said above, succeeding at ensuring compliance in large firms is complex and with the growth of Cloud Computing technologies, it becomes harder. Cloud Computing can be described as the provision of IT services via the Internet to deliver faster, flexible resources and easy scaling (Figure 1.4). Usually, we only pay for the Cloud services we use allowing us to manage our infrastructure more efficiently and satisfy our needs thanks to scaling bringing several benefits:

Cost Cloud Computing remove the cost of having and supporting

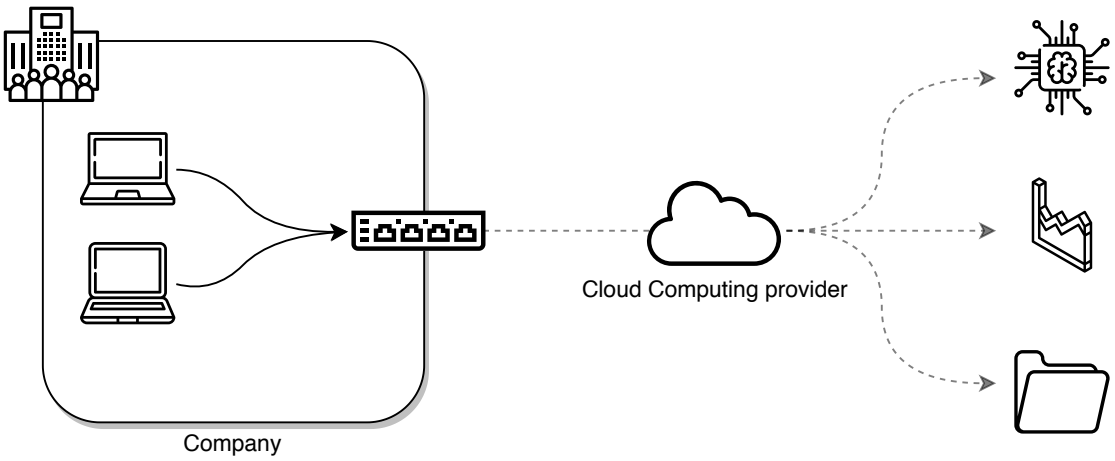


Figure 1.4: Here, a Cloud Computing provides computing, analytics and storage services to a company while the latter has no more infrastructure at all.

infrastructure: physical servers, energy cost, cooling, human resources.

Speed Most Cloud Computing services are provided on demand. Huge computation resources can be up to work in minutes with few clicks.

Worldwide scaling Scaling and localization services offered by Cloud Computing allow to follow the needs closely (more computing power, more storage or bandwidth for example).

Productivity Cloud Computing hosts handle the hardware and upgrade the necessary software themselves, allowing us to remove these chores for our company.

Reliability and security Cloud Computing through Service-Level Agreement (SLA) ensures our backups, restoration and support. In addition, they ensure our security features with up-to-date technologies and process.

The technological breakthrough brought by this paradigm is immense, as we can see in Figure 1.5 and affects many sectors such as SAM. Several types of Cloud Computing exists: public, private and hybrid. Public Clouds are hosted by third-party service provider like Microsoft Azure, Amazon Web Services, Google Cloud or Orange Business Services, to name a few. Private Clouds are companies hosted services for internal use. It is generally located in companies' datacenters or can be hosted by third-party but for the company's private use only. Hybrid Clouds is when we use both, using internal

services we provide and using third-party services because we lack them or because we want to quickly scale.

Before entering the details about licensing in the Cloud, it is already important to see the impact of Cloud Computing on Software Asset Management. Like said before, all digital data are potentially critical to ensure compliance because of PURs, but this amount of data just became exponentially bigger. The time and processes needed to collect every bit of information about physical infrastructure around the world is big, but added to virtualization, it becomes quickly overwhelming. Easy connection between hardware and software, for example, no longer exists. Our product can move quickly between different physical servers while being used by more and more people. The life cycle duration shortened, reaching down to minutes making the detection of installation harder.

Focusing on the deployment of software in the Cloud, Cloud Computing brings a lot of complication. The *metrics* were designed for old architectures and, because of that, can behave differently in Cloud. The definition of the processor *metric* for Oracle Database, for example, states that we have to count all the processors a software can use. This definition, in the Cloud, become very dangerous due to virtualization. Indeed, with the possibility of migrating the software all over the Cloud, the number of usable processors multiply quickly. In Figure 1.6, we can see an example of the impact of such *metric* with public prices. The new models, besides bringing fresh *metrics* based

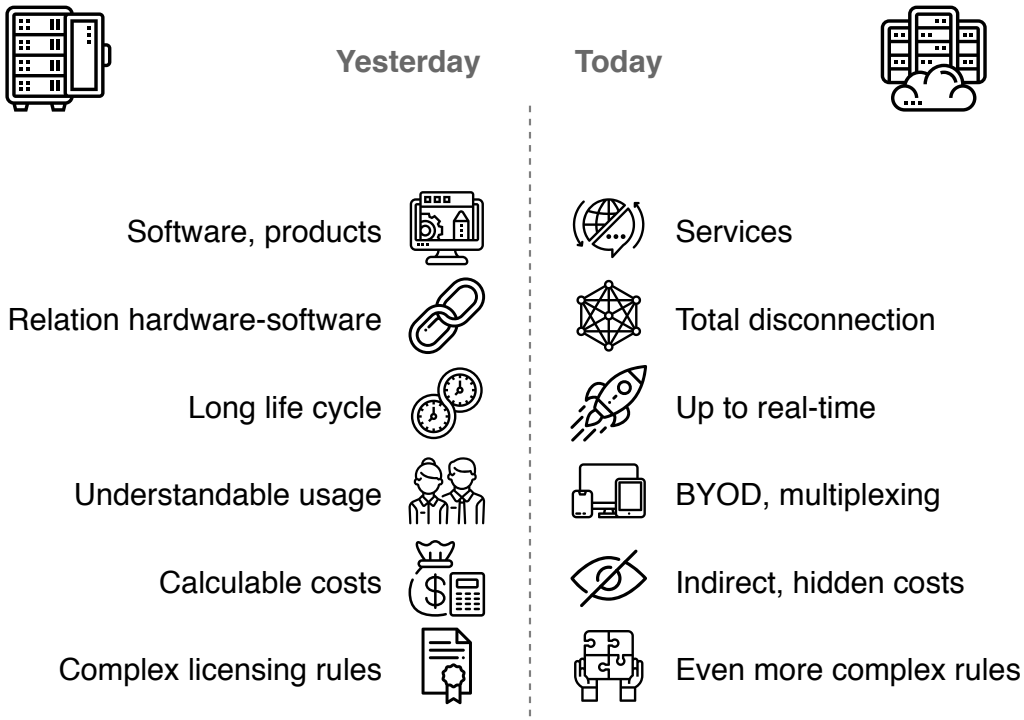


Figure 1.5: Difference between yesterday's infrastructure and today's Cloud computing

on usage and consumption, raise many questions. The use and licensing of proprietary software in public Cloud is a troublesome matter. Two methods exist for the licensing responsibility: Either the third-party host licenses the software and uses a subscription-based or time based licensing for its offer or the one who brings the software brings with it the licenses, called the Bring Your Own License (BYOL) concept. With hybrid Cloud, if a software migrates from one type of Cloud to the other, it can affect the licensing and, therefore, will require compliance check from the software asset managers.

Another complication is that the information we needed to compute the *metrics* are not enough anymore. With the same example of a VMWare virtualized infrastructure from Figure 1.6, depending on the used version, a Virtual Machine (VM) can go from one vCenter to another, multiplying again the number of usable processors. Therefore, we will need information not only on processors but on other software too. Here, a single upgrade of VMWare from version 5.5 to 6.0, even if we have licenses and are compliant for this upgrade, can affect the Oracle Database licenses.

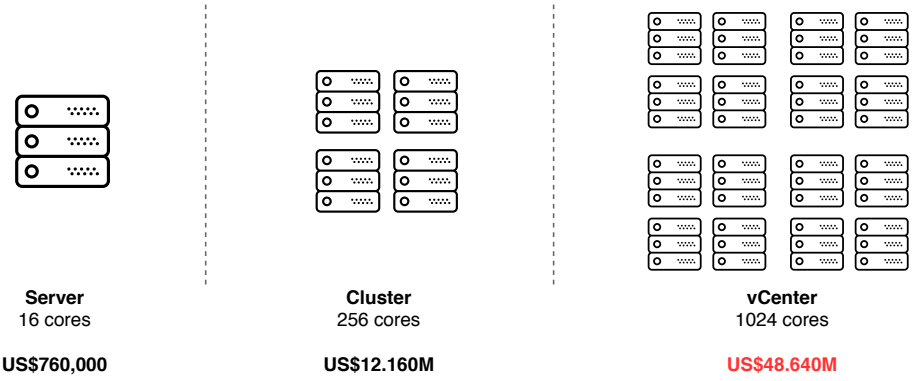


Figure 1.6: Here, using a VMWare virtualized infrastructure and with a corefactor of 1 for all processors, we can see that the impact of using a single Oracle Database on different environments have an immense impact on the bill.

Even the notion of product and software changed because of some alternative models: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) which are described in Figure 1.7. SaaS providers will have to license the software they provide and indirectly make the user pay with another kind of subscription-based license. PaaS and IaaS providers, on the contrary, let the user install its own software but, the licensing become difficult. As the user may not know the infrastructure details or the virtualization used, the *metric* computation is maybe not doable anymore. Here, a good knowledge of what the provider offers for licensing issue is required and the latter has to be carefully followed (for example, AWS [8] made a white paper about Oracle Database licensing).

Some could say that Open-Source Software (OSS) could solve this problem: no more *metric*, no more paid licenses. Indeed, OSS can be a solution if we start with it and stick to it. Problem is, OSS does not have any insurance, meaning that it will not

ensure support in the middle of the night because our billing software crashed. If we need specific features, OSS will be slow to implement it compared to contractualized company we bought it from. It will take time to implement security patches compared to an editor offering support services such as RedHat, for example. As soon as we have to ensure quality, support and performance to our client, we will need the same thing from our software provider. Besides, if we already use some proprietary software, it will be difficult to switch. For example, around 2000, Amazon started looking to replace Oracle Database to open-source alternatives and finally achieved it late 2019 [9]. In summary, OSS can be an answer, but as it is not the ultimate answer yet, we will focus on enhancing the situation with proprietary software.

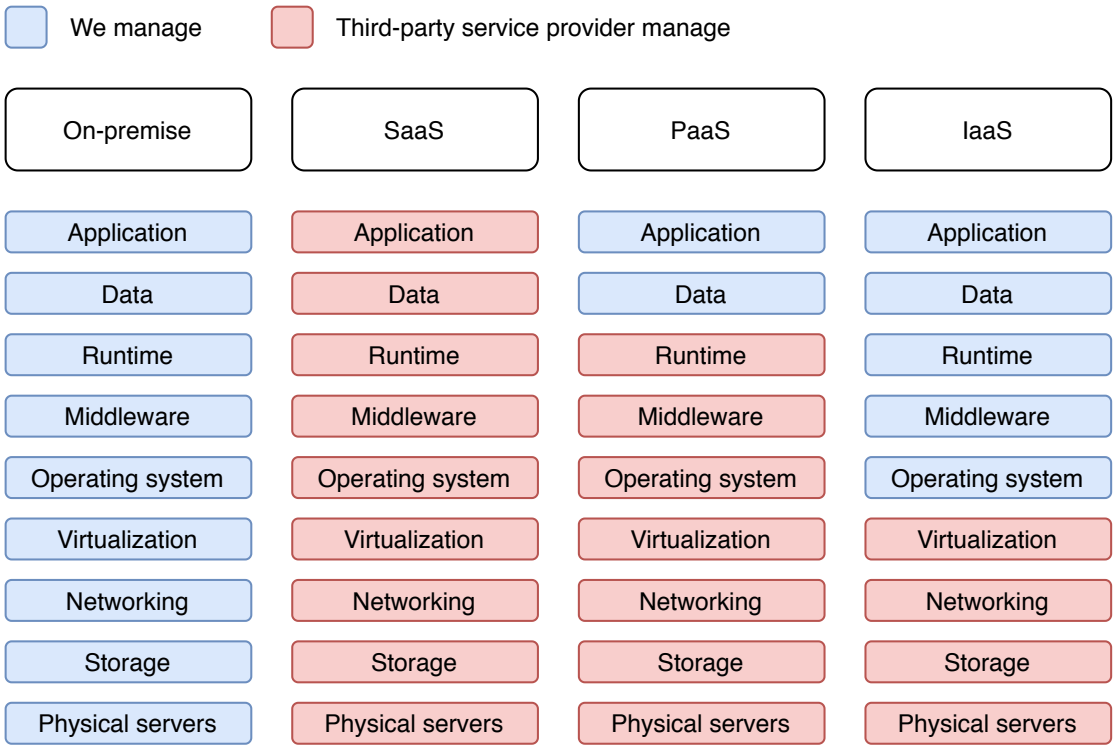


Figure 1.7: Difference between SaaS, PaaS and IaaS: Microsoft Office 365 and Google Docs are SaaS, Microsoft Azure can provide PaaS services and Amazon Web Services provide IaaS services.

Modern Software Asset Management is not adapted to such infrastructure firstly because of the enormous amount of data and computation needed and secondly because of the compliance checking process actually in place. As the *metrics* are based on usage, the *metrics* computation is made retroactively (after the use of the software) and, therefore, a noncompliant situation is detected too late. In addition, the impact of a single modification on the software park is detected afterwards, making the cancelation of such change very difficult. We will tackle this problem by proposing some advances in

Software Asset Management to ensure compliance before the deployment. After talking about the State of the Art, we will start by introducing a standard to facilitate the understanding of contractual *metrics* and automatizes it. Then, we will present an algorithm which automatizes the *metric* selection and impact management for any software installation before proposing a new deployment heuristic for the Cloud. Finally, we will submit a multi-software deployment algorithm based on the proposed heuristic.

1.3 State of the Art

In the context of this thesis, we will use the International Organization for Standardisation (ISO) description of a software asset [10]: "all or part of the program which process or support the processing of digital information and has potential or actual value to an organization". As such, we can find executable software (Oracle Database, Microsoft Windows) and non-executable software (fonts, spell-checker dictionaries). It has been further described in the literature as "all software objects or artifacts along with processes used to define, create, develop, acquire, and evolve or maintain software." [11] as well as a "fixed asset that contribute to a company's productive capacity directly or indirectly." [12]. We will also place ourselves in the context of firms using proprietary software and will not address the open-source case. Such companies can be of very different sizes, from the smallest to a major US insurance company which took 18 months to produce an inventory of one third of their products [13]. Digitalization of companies are the key to unlock competitive advantages by doing things better, faster, and cheaper than the competition. Businesses will use software to promote quality and consistency to improve the customer relationship [14, 15] and, therefore, are critical to them.

The usual industrial question "Buy or Build" can be easily translated in the digital world with two kinds of software: Third-Party Software (TPS) or Custom Developed Software (CDS). A company must choose between making its own custom software that will fit its needs perfectly or using a third-party software. The second choice is special because TPS are composed of Open-Source Software (OSS) and Commercial Off-The-Shelf software (COTS). A benchmark about the security of the different types [16] drew the conclusion that while companies are using TPS in greater quantities, this kind of software present the most security risks. It is mainly explained by the fact that companies trust editors and fail to check software quality before the integration. The problem is not solved with OSS as explained earlier, as anyone can propose code, the same failure of quality check leads to IT risks [17] and critical situation like the LeftPad repository huge chaos [18].

One of the first publications about licensing was devoted to the study of automated *metrics* of software, the assessment of CIM repository and management of software assets [19]. Real Software Asset Management considerations started in 1999 when a study about the model and identification of software was proposed [20]. This study stated that SAM can mitigate technical, legal, managerial, financial and ethical risks in organizations. Later, in 2005, the need for a framework for control of software assets throughout their life-cycle to ensure long-term and efficient management has been showed [11]. In

2011, a proposition to combine IT, processes and SAM was offered and revolved around four points:

- Being able to discover software
- Being able to make precise inventories of licenses and the infrastructure
- Implementing contract management
- Producing reports about readiness towards compliance and verification

In 2017, a patent [21] proposed tools for discovering and collecting information on instances of software used in monitored environment. In the same time, the Cloud was added in SAM considerations in a review of existing SAM tools and a new SAM model for Cloud architectures [22].

Despite being a modern key to enable digitalization in companies, IT managers fail to address SAM issues and ignore the necessity of having a proper framework for compliance verification and vulnerability analysis [23]. Less than 20 percents of companies effectively use SAM [24] and while 65% of enterprises are audited yearly (up to 23% of them were audited three times or more on the same year), only 29% have an automatized monitoring of their systems and 25% of them have no monitoring at all [25]. Such lack of SAM could result in huge expenses for companies [11, 26] and can sometimes lead to severe trouble like the 2000 year problem that leaded to enormous costs to firms that didn't have SAM inventory databases [27, 28]. While business, for security purposes, setted up centralized security policy and denied to their employees the right to install software on the professional work station, the BYOD paradigm tends to reverse this situation and let people use as they see fit their devices leading to IT risks [15, 29]. Academic field showed the heavy financial risks in case of SLA breach [30] and backed up the Flexera Study [25] by showing the lack of SAM in companies [14, 23, 31].

All this academic history show clearly the need for Software Asset Management in enterprises. Coupled with the rise of Cloud technologies and their impacts on this SAM, which is already fragile in most companies, it becomes obvious that we need new process and above all an automatization of thoses processes to avoid human error and answer the velocity and size of the data induced by such Clouded platforms. Modern SAM date back to another era where contracts and definition are defined on papers, where compliance computations are made thanks to spreadsheets, and where asking a full inventory of software assets can take up to years. This SAM requires an update to handle new emerging, fast, data-intensive applications.

Chapter 2

Metric definition problem

Contents

As showed in the introduction, mostly often-used 'traditional' licensing models (such as a number of cores, CPUs, allocated physical resources) bind software deployments to physical infrastructures or hardware features (ownership, geographical restrictions, installations, etc.). This binds between IT environments and software licenses are limiting usage and capacity, especially when migrating from traditional IT models to flexible Cloud infrastructure. Considering the most commonly used *metrics* (processor, devices, user, access), we can summarize some major risks moving these licenses schemes to Cloud infrastructures. Bound to the processor capacity: these *metric*, especially in virtualized environments are often complex, slightly different from one to another depending on the editor. Keeping track of the proper amount of processor license counts and capacity levels typically requires deployment of advanced monitoring systems. Moving to the Cloud, monitoring systems to track processors counts and capacity levels in IaaS can be more challenging because of compatibility, security and network issues. Bound to devices: software can be accessed and used via multiple devices (virtual/physical); thus, keeping track of licensable devices can be challenging. Bound to user: usage rights for each user role are tailored in software license agreements. Access to usage rights can hardly be technically restricted and are difficult to report and translate into licensed roles when Cloud demand real-time visibility on user's usage right assignments. Bound to In-direct Access : these licensing rules often call for all interactions between software and human users either directly, through a named account, or indirectly through a shared account or third-party application account, to be fully licensed.

Effective Software Asset Management (SAM) results in the ability to have accurate and complete view of software assets entitlements that are owned, deployed and used. It consists first in intercepting all software license sales and purchases. It encompasses knowledge on software and its entitlements identification. Then, it consists

in intercepting each software installation bound workflow configuration, usage *metrics* and consumed resources workflow. Theses different views have to match. Unless legally armed, the definition of *metrics* is a bit at the discretion of publishers, which can be problematic in many cases. In this chapter, we introduce a modeling of the *metric* that will bring some enhancement:

- Automated algorithm can process a structured *metric*
- It unifies the understanding of the *metric*
- Affects are clearly visible

This model is a representation of the licensing context of IT environment: it contains the physical and virtual representation of the Cloud as well as the representation of third-party data (employees, industrial economic figures, contractual data, etc.). This global context allows us to apply each *metric* with the right data and also allows us to verify our ability to 'assume' these same *metrics*.

2.1 Modelization

The data model present in Figure 2.1 represents the computing environment in the form of a graph where each node is an important data set. We designed the model to be as generic as possible to answer any kind of *metric*.

It is composed of the following nodes:

entity An $\langle Entity \rangle$ describes a group of people working on a specific domain (strongly related to business entities) and is used to identify the membership and responsibilities related to other nodes. The entity system is hierarchical, which implies that entities are represented in the form of a tree and therefore one $\langle Entity \rangle$ may be related to another. Each $\langle Entity \rangle$ can sign contracts (link to $\langle Contract \rangle$). Besides, they are part of a $\langle Project \rangle$ and have multiple relation to the $\langle Equipment \rangle$ node: they can use an $\langle Equipment \rangle$, host it, or ensure the support of it.

publisher A $\langle Publisher \rangle$ is the seller of the product (not necessarily the developer of the product). It can be an online platform, a physical reseller or an internal entity of the company. This node is linked to a $\langle Contract \rangle$ because we want to keep a record of where we have contracted the product.

editor An $\langle Editor \rangle$ represents the creator/legal owner of a product such as Oracle for the database of the same name or Microsoft for Windows. The name of the editor is the very name of the company to which the software belongs and not the parent company: for example, for Linux Red Hat the editor is RedHat and not IBM. It is important for the case of selling and buying companies, if another company buy RedHat then we have to know which

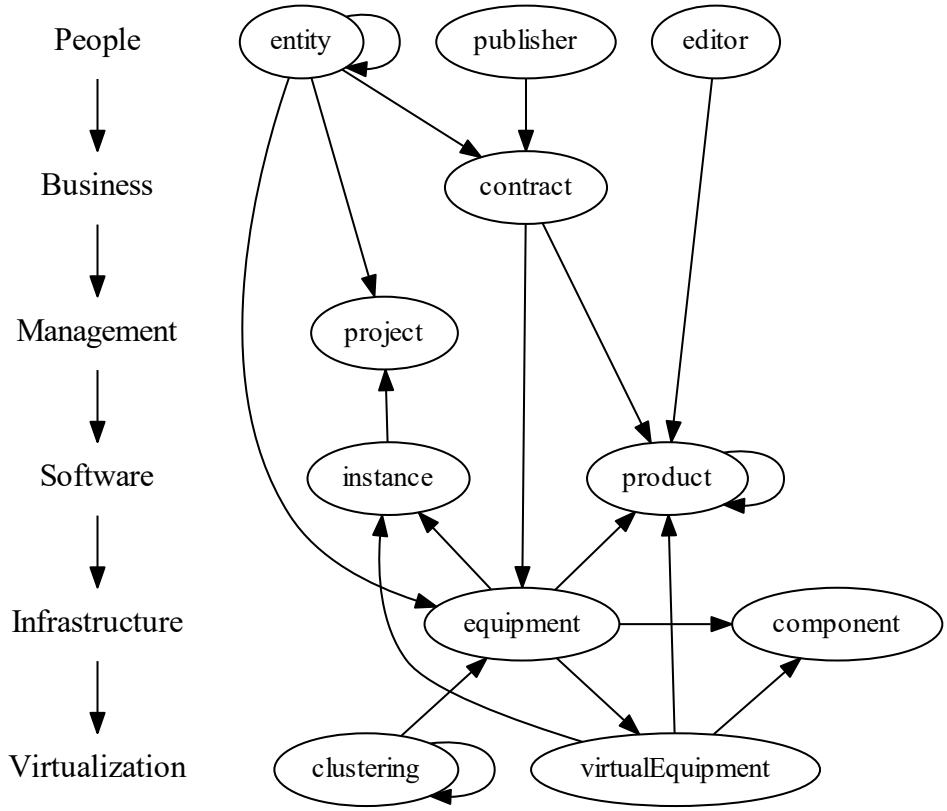


Figure 2.1: Graph model of a *metric*

products and contracts are impacted, which would be impossible if we tagged IBM. An ⟨*Editor*⟩ is linked to the ⟨*Product*⟩ node for obvious reasons.

These three nodes compose the *People* layer that represent all human data interfering with the computing environment. This layer is extremely volatile and the data have higher risk of being outdated or inventoried too late. Therefore, data coming from this layer have to be overvalued to keep a safe distance from the noncompliance zone.

contract A ⟨*Contract*⟩ represents the rights gained at the time of purchase. A contract can cover distinct types of ⟨*Products*⟩ such as a purely software product, fully assembled equipment, operating system or even an ⟨*Equipment*⟩. This contract specifies what has been contracted on our physical and virtual IT assets.

project A ⟨*Project*⟩ is a high-level representation of a digital needing appli-

cation like an application composed of multiple software or a fleet management of autonomous cars for example.

instance The $\langle Instance \rangle$ node represents an instance of a project both in environment (development, production, test, etc.) and in multi-distribution (several instances for redundancy, for example). This allows the machines to be separated by environment or role. Each $\langle Instance \rangle$ contribute to a specific $\langle Project \rangle$.

product A $\langle Product \rangle$ represents any piece of software (executable or not) we have to follow. This node contains identification attributes such as the Stock Keeping Unit (SKU) for example but also name, version, edition. Any $\langle Product \rangle$ can contain other $\langle Product \rangle$ called 'options'. Like this, we can represent enabled options with self version and edition. This node is the centerpiece of this model.

equipment An $\langle Equipment \rangle$ represents any "built" physical material (i.e. excluding components). It can be a server, a router, a phone or a robot if, for example, we are managing a warehouse. Each equipment is composed of different $\langle Components \rangle$ and can be layered in several $\langle virtualEquipment \rangle$ (such as a Virtual Machine or a Docker for a server but it can also be a model of a learning machine for example). Each $\langle Equipment \rangle$ is linked to the $\langle Products \rangle$ installed on it and contributes to one or different $\langle Instances \rangle$.

component A $\langle Component \rangle$ represents a physical piece that composes $\langle Equipment \rangle$ and $\langle virtualEquipment \rangle$. These include processors, memory sticks, network cards, power supplies, and many others. Each $\langle Component \rangle$ has its own attributes and identification that are defined in the underlying dataset. The difference between $\langle Equipment \rangle$ and $\langle Component \rangle$ is that this dataset is static, as attributes for a specific processor will not change for example. We can therefore use the $\langle Component \rangle$ as a library of pieces.

clustering $\langle Clustering \rangle$ represents a virtual grouping of several devices such as the VMware representation of a Cloud (Datacenter, vCenter, Cluster, Servers) but also any other type of grouping. Like the VMware model, this clustering is recursive and each node carries information about its layer of grouping. Obviously the last node is connected to all the equipment it contains. This representation also allows higher level clustering nodes to be connected directly to the equipment as if a datacenter had several vCenters but also a room full of robots. We should therefore have direct links from the datacenter (higher-level clustering node) to the robotic equipment in it.

vEquipment The $\langle virtualEquipment \rangle$ node represents each virtual device that can be contained in a device. As mentioned above, it can be of any shape from a simple Docker container to machine learning models or even a very low level machine virtualization like Enterprise 10000 systems. This virtual equipment can be connected to several components, because a virtual machine is assigned one or more CPUs and a quantity of memory, for example. As an equipment, several products can be installed on it and an OS if appropriate. Finally, this virtual equipment contributes to an instance of a project.

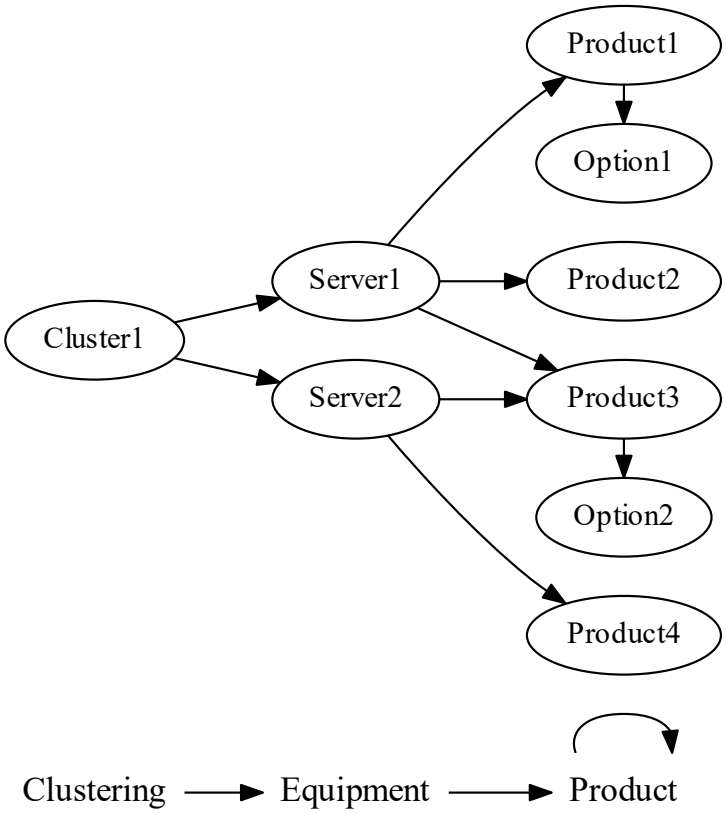


Figure 2.2: Minimal example of relation equipment-product

This model is a flatten view of the bigger graph containing all of our data. The $\langle Equipment \rangle$ node represent all nodes representing an equipment and all those subnodes are connected to $\langle Product \rangle$ node. A minimal example is represented in Figure 2.2 and

a realistic one with only the ⟨*Equipment*⟩, ⟨*vEquipment*⟩ and ⟨*Product*⟩ for 100 servers in Figure 2.3. The underlying graph will be called the '*subgraph*' for the rest of this chapter.

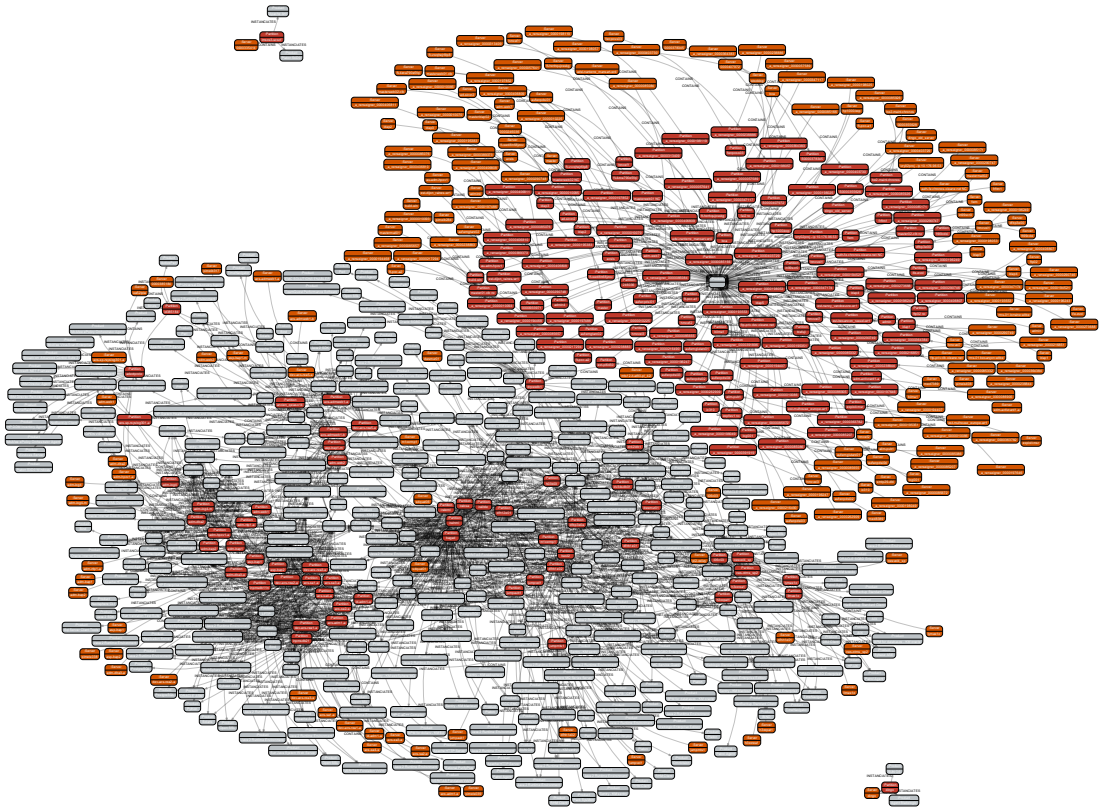


Figure 2.3: Relations between ⟨*Equipment*⟩, ⟨*vEquipment*⟩ and ⟨*Product*⟩ for 100 servers.

Additionally, we proposed a simple language to request '*subgraph*' data through the data model called License *metric* Language (LML). This language is used to represent *metric* formulas and is applied on the different nodes described above. We can see some examples of its requests in Figure 2.4. LML is composed of basic operations and is represented thanks to commonly used tokens:

constants

```
CONSTANT = 0[xX][a-fA-F0-9]+ | [0-9]+ | [0-9]+[Ee][+-]?[0-9]+
         | [0-9]*\.[0-9]+([Ee][+-]?[0-9])?
         | [0-9]+\.[0-9]*([Ee][+-]?[0-9])?
STRING_LITERAL = [a-zA-Z_]?\'(\\.|[^\\\'])+\'
                 | [a-zA-Z_]?"(\\.|[^\\"])*"
```

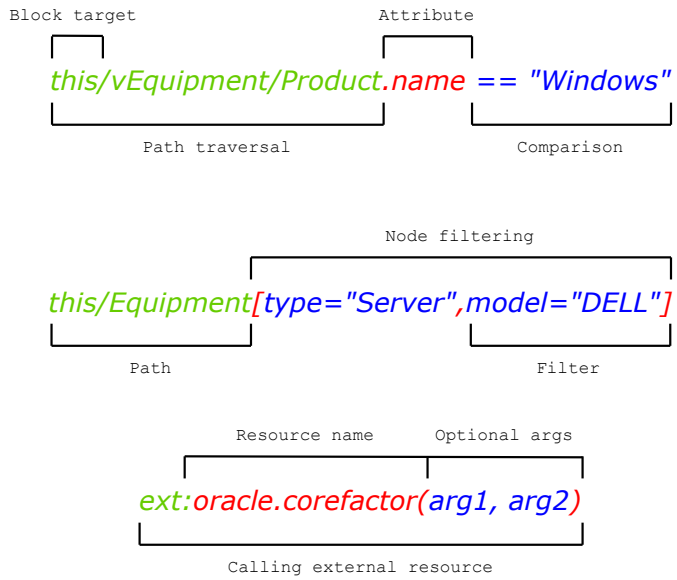


Figure 2.4: Features examples of LML

Comparisons

```
LE_OP = <=
GE_OP = >=
EQ_OP = ==
NE_OP = !=
AND = &&
OR = \|\|
INF = <
SUP = >
```

Operators

```
RIGHT_OP = >>
LEFT_OP = <<
AND_OP = &
OR_OP = \|
NEG = !
SUB = -
```

```
ADD = \+
MUL = \*
DIV = //
MOD = %
POW = \^
```

Assignment and parenthesis

```
ASSIGN = =
L_PAREN = \(
R_PAREN = \)
```

The features brought by LML and showed in Figure 2.4 are path traversal, filtering and external resources. They require specific defined tokens:

```
L_BRACKET = \[
R_BRACKET = \]
DOT = \.
COMMA = ,
SUBNODE = /
EXT = ext
THIS = this
ID = [a-zA-Z_]([a-zA-Z_]|[0-9])*
```

For clarity purposes, the full LML tokens and grammar can be found in Appendix.A (page 79). Each statement of LML is called a request, they allow the user to manipulate the data to represent any *metric*. The request is represented with the following grammar:

⟨*statement*⟩ ::= ⟨*mutation*⟩
| ⟨*comparison*⟩
| ⟨*function*⟩
| ⟨*attr*⟩
| ⟨*operation*⟩

The *mutation* allows to modify data during the computation of the *metric* by assigning the result of an operation or a function to an attribute:

⟨*mutation*⟩ ::= ⟨*attr*⟩ ASSIGN ⟨*operation*⟩
| ⟨*attr*⟩ ASSIGN ⟨*function*⟩

The *operation* is defined as the result of the application of one or multiple operators on values or functions:

⟨*operation*⟩ ::= ⟨*operation_arg*⟩ ⟨*op*⟩ ⟨*operation_arg*⟩

⟨*operation_long*⟩ ::= ⟨*operation*⟩ ⟨*op*⟩ ⟨*operation_arg*⟩

⟨*op*⟩ ::= ADD
| MUL
| DIV
| SUB
| POW
| POW
| LEFT_OP
| RIGHT_OP
| OR_OP
| AND_OP
| MOD

⟨*operation_arg*⟩ ::= ⟨*attr*⟩
| ⟨*val*⟩
| ⟨*function*⟩

⟨*val*⟩ ::= STRING_LITERAL
| CONSTANT

All externally defined functions are accessed thanks to the EXT token. The function is then accessed by its ID before inputing all the parameters:

⟨*function*⟩ ::= EXT DOT ⟨*function_external_long*⟩
| ID L_PAREN ⟨*func_args*⟩ R_PAREN
| ID L_PAREN R_PAREN

⟨*function_external_long*⟩ ::= ⟨*function*⟩
| ID DOT ⟨*function_external_long*⟩

⟨*func_args*⟩ ::= ⟨*possible_arg*⟩
| ⟨*possible_arg*⟩ COMMA ⟨*func_args*⟩

⟨*possible_arg*⟩ ::= ⟨*attr*⟩
| ⟨*val*⟩
| ⟨*path*⟩
| ⟨*function*⟩

The *comparison* grammar is based on comparison operators that can be chained:

⟨*comparison*⟩ ::= ⟨*comparison_set*⟩
| ⟨*comparison_set*⟩ AND ⟨*comparison*⟩
| ⟨*comparison_set*⟩ OR ⟨*comparison*⟩

⟨*comparison_set*⟩ ::= ⟨*comparison_arg*⟩ ⟨*comp_op*⟩ ⟨*comparison_arg*⟩

⟨*comparison_arg*⟩ ::= ⟨*attr*⟩
| ⟨*val*⟩
| ⟨*function*⟩
| ⟨*operation*⟩

⟨*comp_op*⟩ ::= LE_OP
| GE_OP
| EQ_OP
| NE_OP
| INF
| SUP

The path traversal and filtering features of LML are defined with the following grammar:

⟨*attr*⟩ ::= ⟨*path*⟩ DOT ID
| NEG ⟨*path*⟩ DOT ID

⟨*path*⟩ ::= ⟨*node*⟩
| ⟨*path*⟩ SUBNODE ⟨*node*⟩

⟨*node*⟩ ::= ID
| THIS
| THIS ⟨*filter*⟩
| ID ⟨*filter*⟩

⟨*filter*⟩ ::= L_BRACKET ⟨*filter_comparison_args*⟩ R_BRACKET

⟨*filter_comparison_args*⟩ ::= ⟨*filter_comparison*⟩
| ⟨*filter_comparison*⟩ COMMA ⟨*filter_comparison_args*⟩

⟨*filter_comparison*⟩ ::= ⟨*filter_comparison_set*⟩
| ⟨*filter_comparison_set*⟩ OR ⟨*filter_comparison*⟩

⟨*filter_comparison_set*⟩ ::= ID ⟨*comp_op_filter*⟩ ⟨*val*⟩

⟨*comp_op_filter*⟩ ::= ⟨*comp_op*⟩
| ASSIGN

This allows the user to target a node or an attribute and filter the '*subgraph*' nodes based on their data. We can see in Figure 2.5 the finite state machine of the LML grammar.

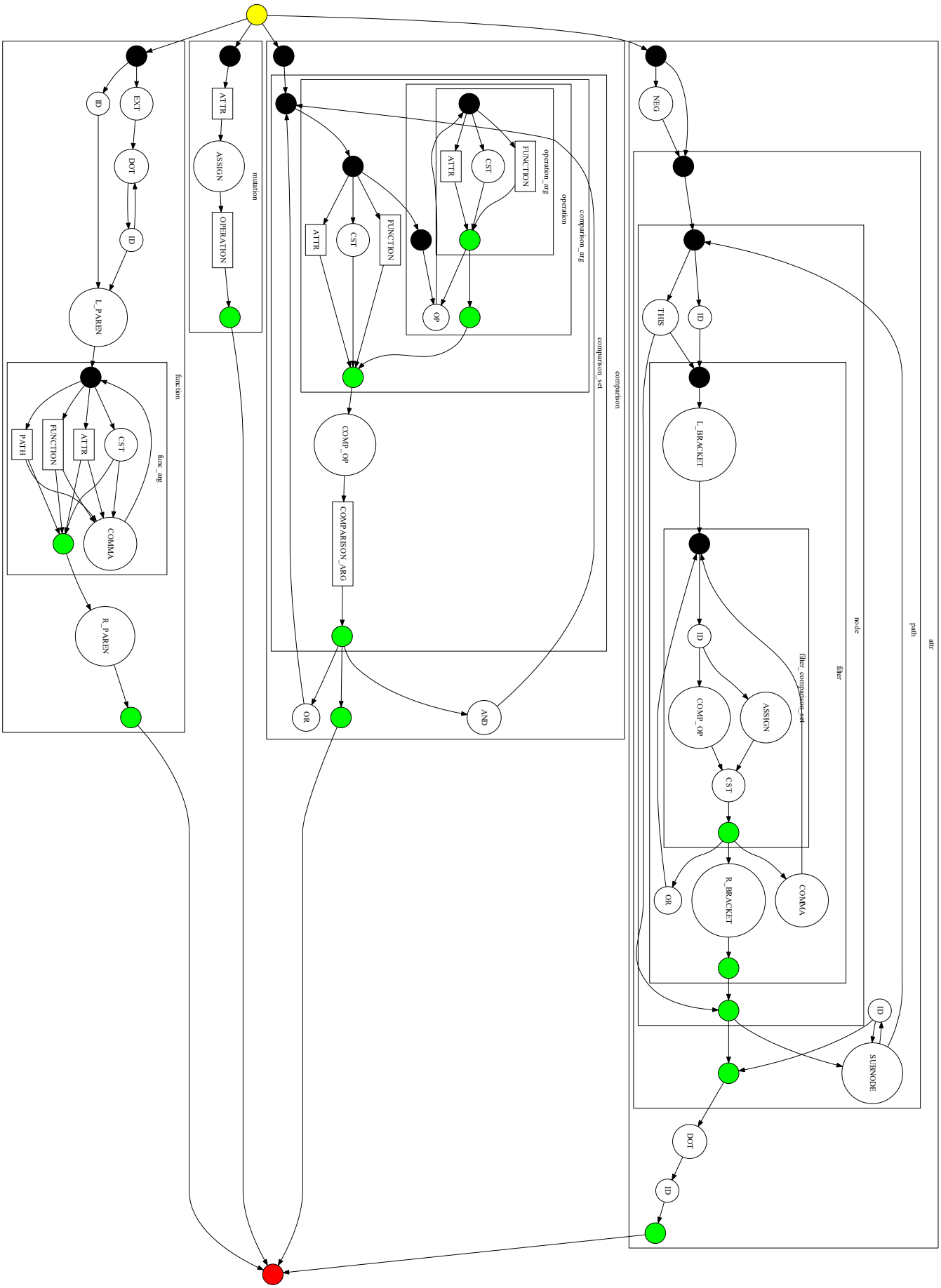


Figure 2.5: Finite-State Machine of LML.

The principle of the *metric* model is that we can tag nodes with LML languages. A tag comprises a specific keyword followed by the LML code. We can see in Figure 2.6 an example of tag on the ⟨*Equipment*⟩ node. To describe a *metric*, it is necessary to tag each adapted node using the following set of keywords:

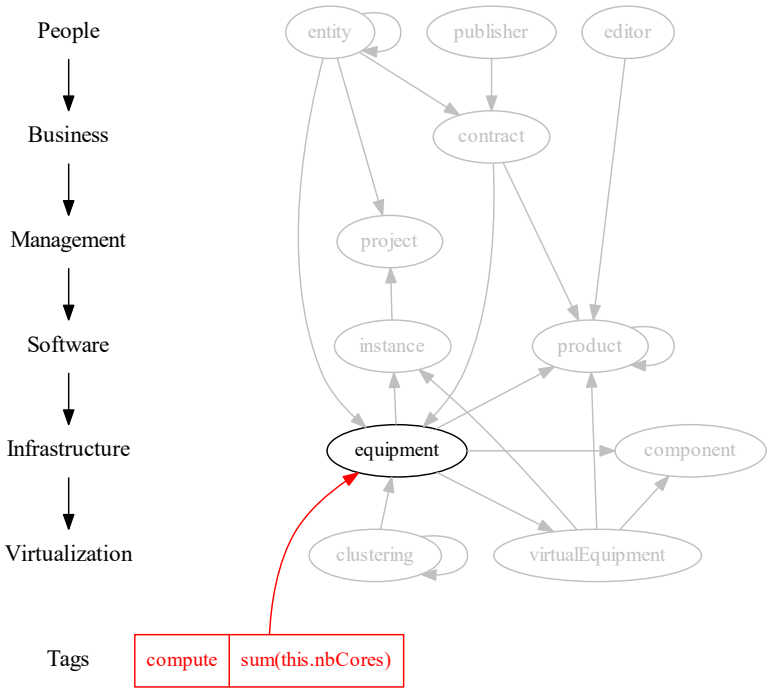


Figure 2.6: Basic example of node tagging with LML

filter The filter keyword allows us to filter nodes depending on their data, for example, to keep only the equipment in production:

$$this.environment == \text{“}Production\text{”}$$

This filter is active throughout the processing of the active block node. This allows different filters to apply to the same type of node using multiple blocks definitions in the same model:

```
Equipment:
    filter:
        this.environment == 'Production'
    *Process filtered nodes in production environment*
```

```
Equipment:
    filter:
        this.environment == 'Development'
    *Process filtered nodes in development environment*
```

stop The keyword stop allows to express an impossibility to calculate the *metric* thanks to a logical expression as well. For example, if a *metric* should only be used in a student environment, the calculation should be stopped if we are in a professional context. This allows for example to quickly discover a noncompliance with a *metric* during use or before a contractualization.

mutate Mutate allows us to change node attributes by following a calculation formula. With Oracle's processor *metric*, we must multiply the number of cores by a corefactor [5], which is done using the following expression:

$$this.nbcores = this.nbcores * ext : oracle.corefactor(...)$$

During a block processing, all modified data are stored in a cache so that the original data stay untouched.

gatherBy gatherBy is used for aggregation functions targeting a node or entity. For example, to calculate the number of cores required with Oracle's processor *metric*, it is actually necessary to count all the cores present in a vCenter so in our case it would be necessary to use the gatherBy keyword targeting the Clustering node representing a vCenter.

compute compute defines the calculation to be performed to get the required number of licenses according to the modeled *metric*. This key must be present in each model and only once! The block containing the compute keyword will be executed last.

A *metric* is therefore defined in blocks describing the processing of underlying data and the computation needed to get the number of licenses. For example, the Microsoft Windows 'per-core' licensing *metric* with Datacenter edition [32] is defined as a one block model:

```
Equipment:
    filter:
        - this/vEquipment/OS.name == "Windows"
    mutate:
        - this.nbLicensedCores = max(
            16,
            max(8 * this.nbCPUs, this.nbCPUs * this/Component[type="CPU"].nbCores)
        )
    compute:
        sum(this.nbLicensedCores)
```

Besides, Figure 2.7 and Figure 2.8 represent the models of the Oracle Database Processor *metric* and RedHat Instance *metric*, respectively.

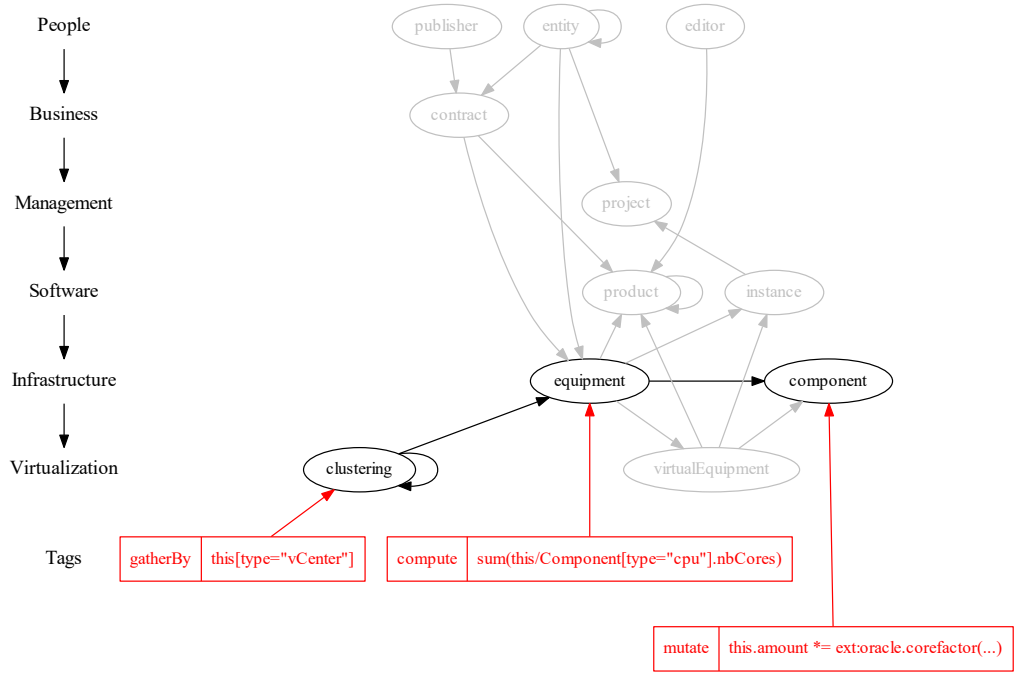


Figure 2.7: Model for Oracle Database Processor *metric*

2.2 Using the model to compute compliance

We can now use this structure *metric* model to automatize compliance checking and optimize license consumption across our software park. We developed an algorithm that compute the number of license from a *metric* structure. This algorithm is separated in 4 phases:

Phase 1: Filtering This step uses the filter keyword in each node. Each LML formula must return be a boolean expression and only the underlying nodes where this expression returns *True* are kept. Each block is tested independently, and then we move to the second step. One example of filtering the underlying nodes is represented in Figure 2.9.

Phase 2: Stopping Here, we will use the keyword stop in the same way as above. If the result of the Boolean formula is *True*, then the calculation of the model cannot be applied. This can be used, for example, if we want to check certain compliance

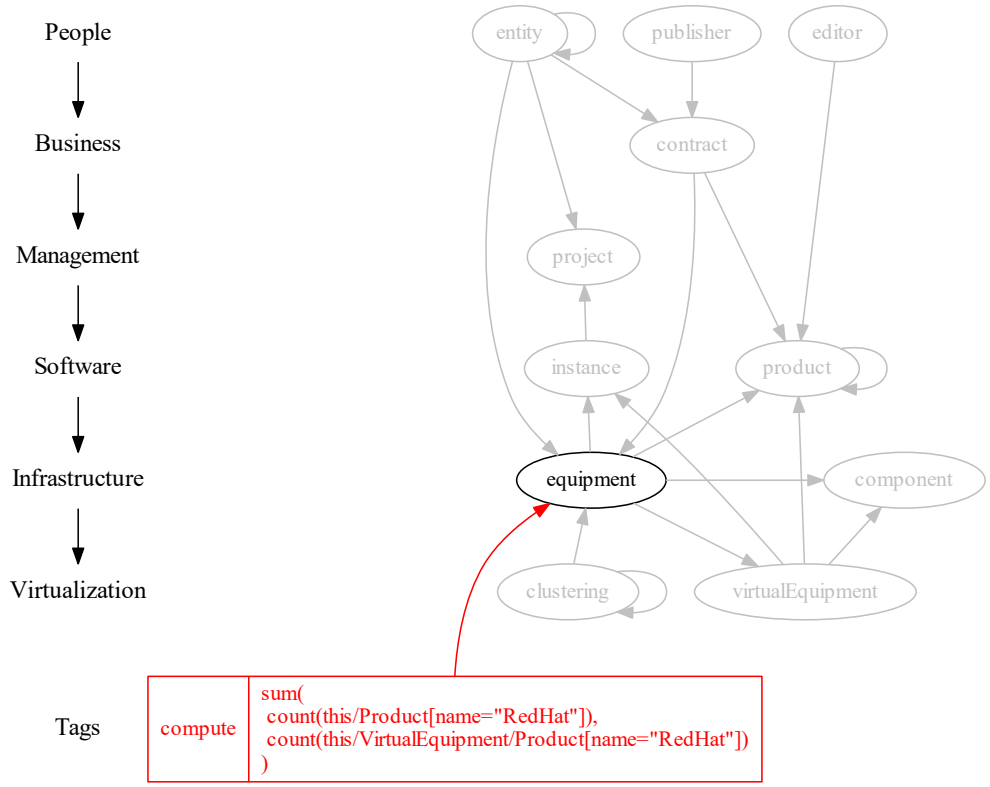


Figure 2.8: Model for RedHat Instance *metric*

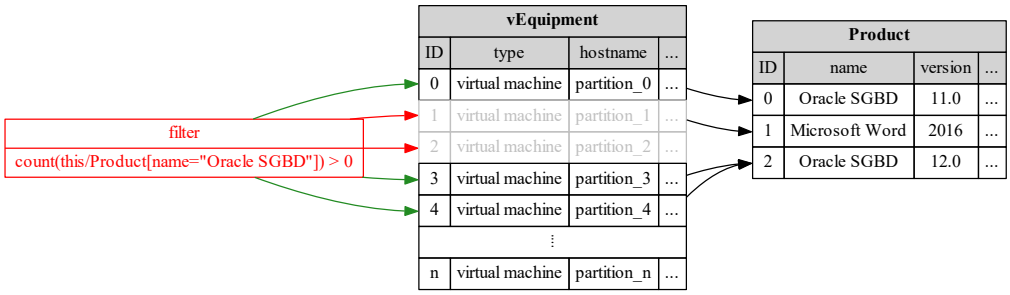


Figure 2.9: Phase 1: Filtering subgraph nodes

issues and respect PURs. This step is used either before deployment or purchase to warn users that this *metric* will place us in a noncompliant state or during the

life cycle to check which contracts are not respected anymore. Figure 2.10 is an example of the stopping phase.

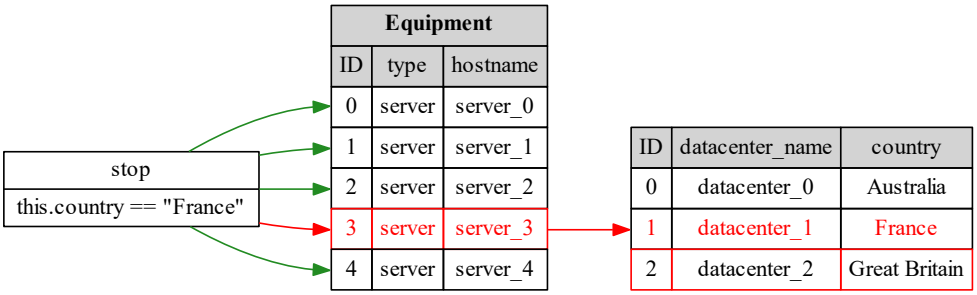


Figure 2.10: Phase 2: Stopping the computation because of noncompliance. Here the server 3 is located in France causing a breach with one of the PURs forbidding this location. This step make it possible to immediately detect a noncompliance and, therefore, indicate that this *metric* is not viable for our use.

Step 3: Mutating This step allows us to change the data of one or more particular nodes. The formula specifies how and this applies to all previously filtered nodes in the same block. Figure 2.11 introduces an example corresponding to the IBM PVU *metric*.

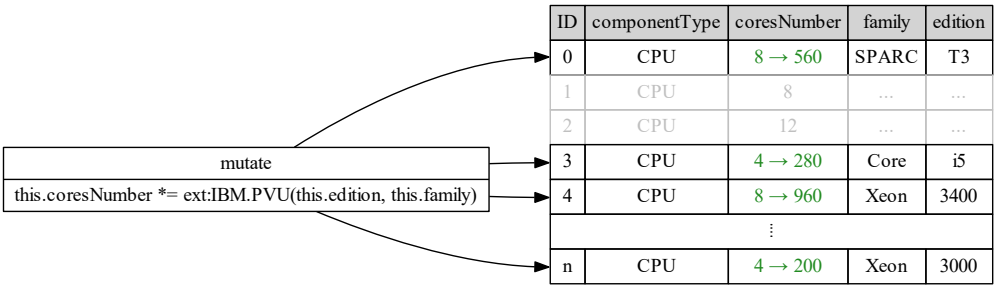


Figure 2.11: Phase 3: Mutating underlying nodes data with IBM PVU example. We can see that the mutated data are only the ones that has been filtered previously within the same block of execution.

Step 4: Computing This last step includes several keywords: gatherBy and compute. An example with the *metric* IBM PVU too is given in Figure 2.12. There can only

be one compute keyword per model and the node containing it will be the last to be executed. During the execution of the compute keyword, only the filtered nodes are affected.

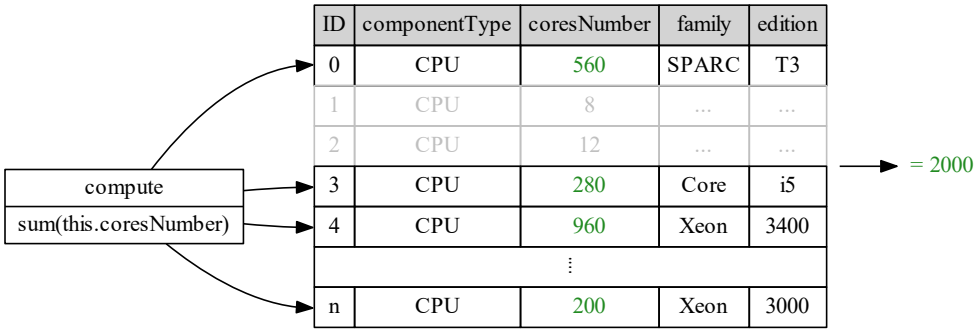


Figure 2.12: Phase 4: Computing the number of licenses

Our implementation uses a graph database to simulate the Cloud environment and run the computations. We chose to use DGraph [33] because it uses a structured query language which has similarities with our structure. To go from the *metric* model to the computation, we use the automaton (Appendix.B) which transform the model into an Abstract Syntax Tree (AST) like in Figure 2.13.

For the evaluation we built a platform that allows us to describe models and then use them on a prototype of a randomly (or not) generated Cloud. This platform allows three things: model writing, computation and visualization of results for model or computation method improvements. The first one is a basic storage of model in a database for further use. The computation generates the Cloud environment and launch the model blocks commands on it before storing the results. The last step, visualization, allow to compare results on different Clouds. We can approve the model with Cloud generation as we can specify exactly the environment and compare with hand computation. We can see in Figure 2.14 the result of the RedHat *metric* on a random Cloud.

As we can get the processing time of the *metric* with this prototype, we compared several *metrics* within hundreds of different Clouds to see the impact of Cloud size and scattering (i.e. more or fewer servers per clusters). We can see in Figure 2.15 and Figure 2.16 that the time needed to compute the *metric* on up to a thousand servers is very low. More importantly, it is linear and does not depend on the data as we computed the *metrics* on hundreds on generated Clouds before taking the average time. This computation have been made on a small laptop equipped with an i3 processor and 4GB of RAM, meaning this calculation doesn't require huge computation power and that we can reduce the time or compute on bigger data sets with a more powerful machine.

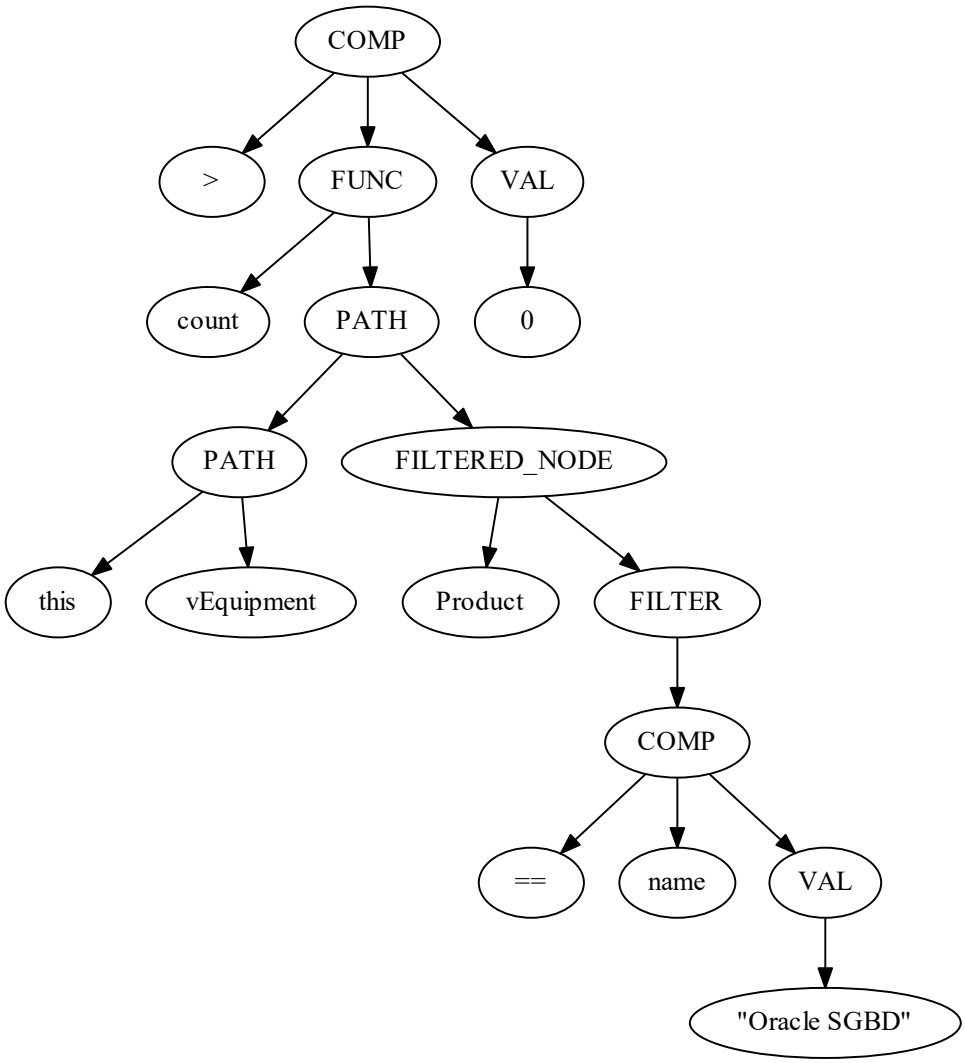


Figure 2.13: Abstract Syntax Tree resulting from the following LML: 'count(this/vEquipment/Product[name=="Oracle SGBD"]) > 0'

The validity of this *metric* model comes from the fact that we manage to represent tens of *metrics* with greater or lesser difficulty. Therefore, we assume that this model can represent any present or future *metric*:

Lemma 1. *The combination of LML and environmental graph representing the metric model presented in this Chapter can represent any present or future metric.*

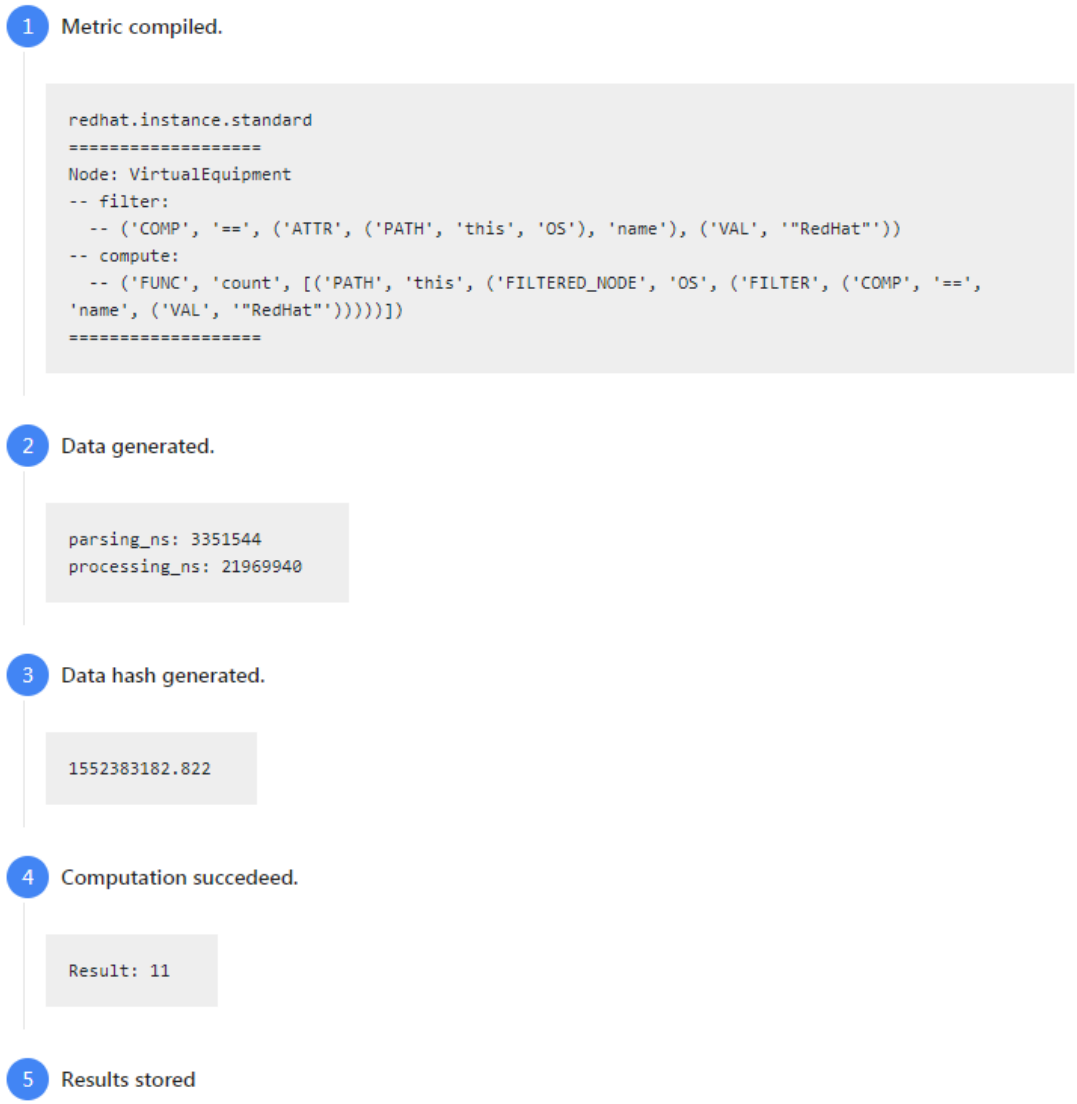


Figure 2.14: Result of LML computation on RedHat instance *metric*

In conclusion, we introduced a Cloud environment representation based on graph coupled with a new language, LML, to model any existing *metric*. We showed that this structure can be used to automatize SAM tasks like compliance checking and that the time needed to compute one model and address a Cloud containing hundreds servers is expressed in tens of milliseconds on a low-level laptop, therefore, being accessible to anyone. One problem remains, if a mistake creeps into a structured *metric* definition then all following computation will output erroneous results irrevocably leading to noncompliance. The solution is to get this model directly from the editor besides the contract

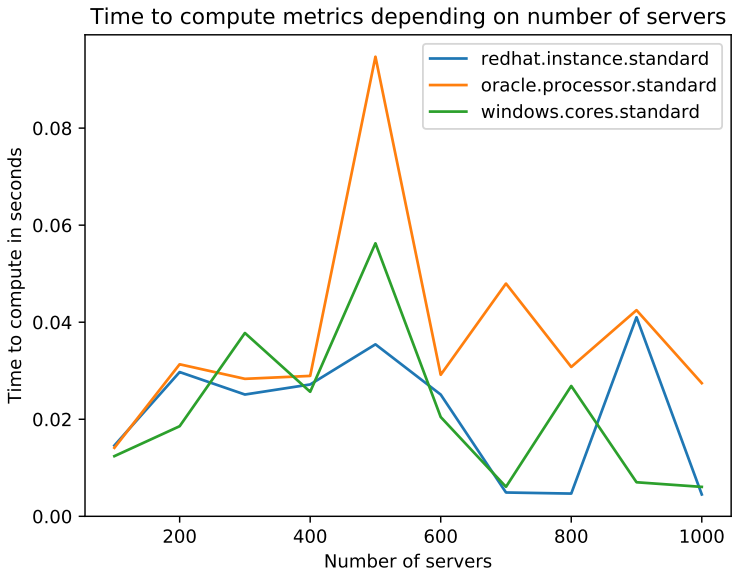


Figure 2.15: Metric model computation time depending on Cloud size

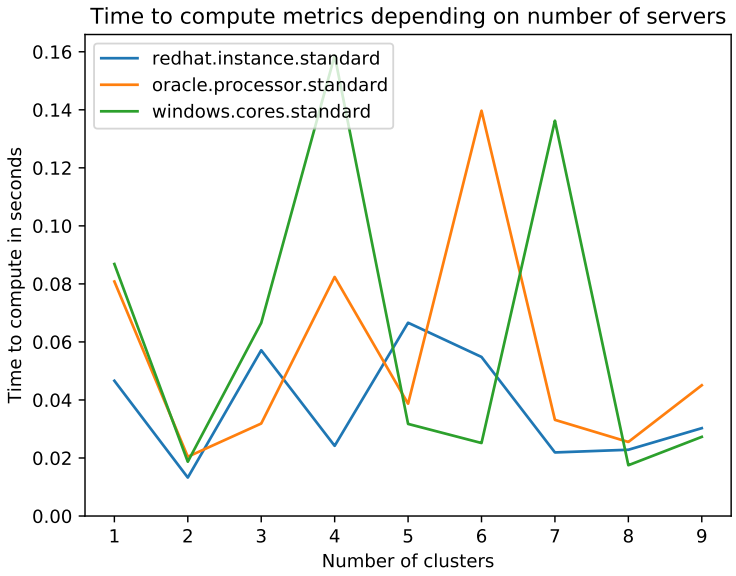


Figure 2.16: Metric model computation time depending on Cloud scattering

and product. Therefore, no model could contain errors, or at least not because of the customer. This can be done in two different ways: First, we can include this model in the ISO 19770-3 [34] which deals with PURs representation but lacks a deeper model of

metric. Secondly, we can extend the Open Digital Rights Language [35] (ODRL) which is a structured language to represent use rights of digital property. This language does not have any *metric* representation but could be used in addition to LML as an exchange standard between editors and customers of license agreements. With such a model, we can now assume that we have a way to compute a *metric* automatically on a given Cloud. These considerations lead to the following Chapter about the automatization and optimization of product deployment on the Cloud.

Chapter 3

Deployment automatization and optimization

Contents

Current architectures handle two ways of deploying products: First, someone installs the product on a platform for further use. This person may or may not have Software Asset Management (SAM) considerations and therefore will maybe break some agreements. As explained in the introduction, some new methods like BYOD introduce more and more risks for SAM. Even if the user of the product knew a little bit about SAM, the amount of data he would have to access is enormous, as well as his job of cross-checking all contracts to see the impact of such installation of other contracts about other products. All the while assuming he has access to all this data. The second way of deploying products is by going through an interface and asking for a product. This platform will then compute the compliance and will authorize or not the deployment of the product. This second way already exists for Cloud orchestrators which try to deploy the product following some rules and optimization like storage, network proximity, and energy consumption. What an orchestrator doesn't have is SAM knowledge and access to all the data needed to ensure compliance. Another problem is the dynamicity of the environment: What happen if we connect new servers to a Cluster while a product is installed on it? We will tackle this issue by proposing a multi-parametric optimization deployment algorithm that ensure compliance, deploy several products at a time.

3.1 First considerations about SAM for deployment

First of all, we will demonstrate the impact of a bad deployment of a product. Considering only SAM perspectives, we want to deploy an Oracle Database somewhere in a Cloud and see the difference between some placements. This Cloud will be separated in Clusters and each Clusters will have a specific number of servers and by extension cores. As we can see in Figure 3.1, we deploy a database on a Cluster with 4 cores leading to the need for 4 licenses to be compliant (we assume that the corefactor [5] is 1). If we install new servers to this Cluster, adding 2 more cores, then we are no longer compliant. Current orchestrators cannot prevent this behaviour so SAM tools have to detect and fix this issue.

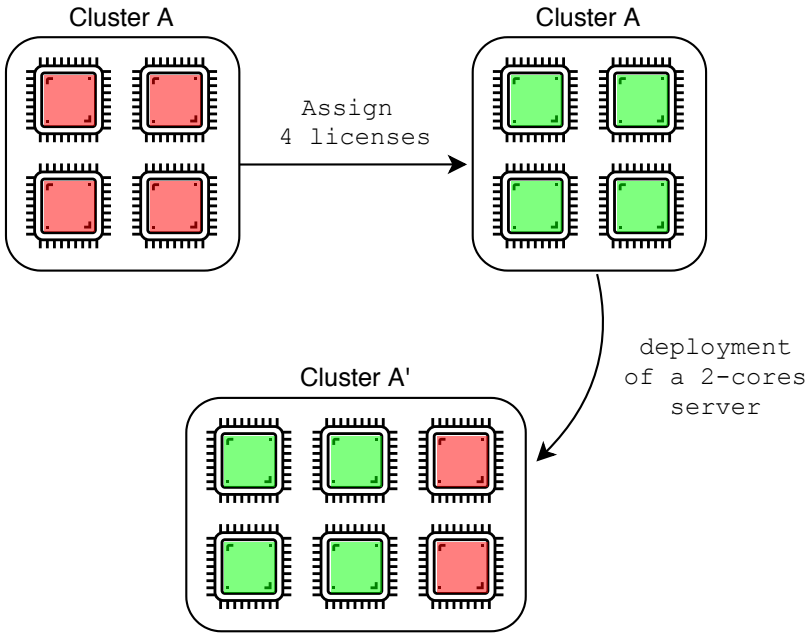


Figure 3.1: Problem of the during life-cycle deployment of a new server: After assigning 4 licenses to this cluster to be compliant, the new server added 2 cores to the Cluster. This become a noncompliant situation until two more licenses are assigned.

While this case is fairly easy to detect and fix, a real case will have to handle hundreds of different *metrics* in a much bigger Cloud. Currently, no algorithm is capable of doing such a thing. Our algorithm is split in three stages: First, before the deployment of a product, we compute the different *metrics* available for this product on our Cloud and choose the best one. The second stage of the algorithm is to check if new resources have been added to the Cloud and if so, compute the compliance on the new environment. Finally, the third stage is the optimization of the environment where we try to move the products on different machines to reduce the overall cost while keeping the compliance.

To show the effect a SAM oriented optimization can have on the price, we will make

a first naïve deployment algorithm for Oracle Database and its two *metrics*. As we have only two *metrics* (defined in Section.1.1), we extract the mathematical formula from them instead of using *metric* models to create a single formula telling which *metric* is the best. Using *metric* models doesn't allow us to do that. The two Oracle Database *metrics* are therefore defined as:

Processor Let n servers, c_i the number of cores of server i, l_i the Cluster number of server i, f_i the corefactor of the CPU of server i, and p_i a boolean stating if there is Oracle Database on server i. L_p, the required number of licenses for *metric* processor to deploy a database on the server j, is:

$$L_p = \sum_{i=1}^{n} (f_i \times c_i) \times (\neg p_i) \times (l_i = l_j) \tag{3.1}$$

With $Cl_j = \sum_{i=1}^{n} (f_i \times c_i) \times (l_j = l_i)$ as the number of cores in Cluster j, we can reduce Equation 3.1 as:

$$L_p = \begin{cases} p_j = 0, Cl_j \\ p_j = 1, 0 \end{cases} = (\neg p_j) \times Cl_j \tag{3.2}$$

Named User Plus Let L_p the number of licenses with processor *metric* for the deployment of the database on server j, n the minimum of processor licenses per user (defined as 25 in the public version of the *metric*), and U the number of users that will access the deployed database. The required number of licenses in NUP *metric*, L_n is:

$$L_n = max(n \times L_p, U) \tag{3.3}$$

We can now cross these two formulas to compute the point where one is better than the other. If we take P_p the price of one processor license and P_n the price of one NUP license then $C_p = L_p \times P_p$ is the price for deploying the database with processor *metric* and $C_n = L_n \times P_n$ is the price for deploying the database with NUP license. We want to deploy X databases on the same cluster so O_n the overall cost of the deployment will be $C_n \times X$ while O_p will be $\sum_{i=1}^{X} C_p = C_p + 0 + 0 + 0 + ... + 0 = C_p$ because of the p_i variable. Therefore, selecting NUP *metric* instead of processor *metric* is defined as:

$$C_n \times X < C_p \tag{3.4}$$

Because of the NUP *metric* definition, two cases appears:

$$C_n \times X < C_p = \begin{cases} n \times L_p > U, n \times L_p \times P_n \times X < L_p \times P_p \\ n \times L_p < U, U \times P_n \times X < L_p \times P_p \end{cases} \tag{3.5}$$

In the first case:

$$n \times L_p \times P_n \times X < L_p \times P_p \tag{3.6}$$

$$n \times P_n \times X < P_p \tag{3.7}$$

$$X < \frac{P_p}{n \times P_n} \tag{3.8}$$

In the second case:

$$U \times P_n \times X < L_p \times P_p \tag{3.9}$$

$$X < \frac{L_p \times P_p}{U \times P_n} \tag{3.10}$$

With the prices of both licenses [6] we can define the constant $\alpha = \frac{P_p}{P_n} = 50$, therefore:

$$C_n \times X < C_p = \begin{cases} n \times L_p > U, X < \frac{\alpha}{n} = 2 \\ n \times L_p < U, X < \frac{L_p \times \alpha}{U} \end{cases} \tag{3.11}$$

With this equation, we can affirm that if $n \times L_p > U$, the processor *metric* is more interesting as soon as we deploy 2 databases or more regardless of the Cloud environment. Depending on this result, we can define the three steps algorithm that will show the benefits of crossing mathematical formulas to optimize the deployment. With the following conventions:

- For a specific Cluster:
  $cluster.c \leftarrow$ Total number of cores for this cluster after application of the corefactor.
  $cluster.ec \leftarrow$ Number of cores remaining (i.e. not used by a database).
  $cluster.nbl_p \leftarrow$ Number of processor licenses assigned to this cluster.
  $cluster.nbl_n \leftarrow$ Number of NUP licenses assigned to this cluster.
  $cluster.nbd_p \leftarrow$ Number of database installed licensed with processor *metric*.
  $cluster.nbd_n \leftarrow$ Number of database installed licensed with NUP *metric*.

- Predefined functions:
  $SmallestCluster(Clusters)$ returns the cluster with minimum number of cores after application of the corefactor.
  $SmallestNonEmptyCluster(Clusters, Cores)$ returns the smallest cluster with at least $Cores$ cores remaining.
  $DeployDatabase(Cluster, NbCores, Metric)$ deploys a database on the specified cluster and remove from the remaining number of cores $NbCores$. The type of license used is specified with $Metric$.
  $ActualProcessorCluster(Clusters)$ returns the cluster with a number of processor licenses above zero and with the greatest number of empty cores. If there is no cluster matching, returns the smallest cluster.

The first step takes an input composed of the database to deploy and the current environment. This step outputs the environment after deployment or a warning if the deployment is not possible. We can see this step in Algorithm 1.

Algorithm 1: Step 1: Deployment

input : Clusters: Array of clusters, CPD: Cores Per Database, U: Number of users
output: Clusters: Array of clusters

$cluster \leftarrow ActualProcessorCluster(Clusters)$;
if $cluster.nbl_p = 0$ **then**
  $cluster \leftarrow SmallestCluster(Clusters)$;
end
else if $cluster.ec > CPD$ **then**
  $DeployDatabase(cluster, CPD, Processor)$;
  return $Clusters$;
end
if $cluster.nbl_p = 0$ ***and*** $cluster.ec > CPD$ **then**
  $C_p \leftarrow cluster.c \times P_p$;
  $C_n \leftarrow max(N \times cluster.c, U) \times P_n$;
  if $C_p < C_n$ **then**
    $DeployDatabase(cluster, CPD, Processor)$;
  else
    $DeployDatabase(cluster, CPD, NUP)$;
  end
else
  $cluster \leftarrow SmallestNonEmptyCluster(Clusters, CPD)$;
  if $cluster = None$ **then**
    return $ERROR$;
  end
  $C_p \leftarrow cluster.c \times P_p$;
  $C_n \leftarrow max(N \times cluster.c, U) \times P_n$;
  if $C_p < C_n$ **then**
    $DeployDatabase(cluster, CPD, Processor)$;
  else
    $DeployDatabase(cluster, CPD, NUP)$;
  end
end
return $Clusters$

The second step check the compliance with removed/added resources. It will raise a warning if the Cloud is not compliant anymore. Following the warning, the user that made the change can either remove/add the resource or assign more licenses. This step is defined in Algorithm 2.

Last step is to optimize the environment by moving products on other clusters if it reduce the overall cost of databases installed in our Cloud. This step is defined in Algorithm 3 using the public data about prices and minimum of processor licenses per

Algorithm 2: Step 2: Cloud compliance check

input : Clusters: Array of clusters with removed/added resources, U: Number of users
output: Boolean: Compliant situation or not
for ***all*** $Clusters$ ***as*** $cluster$ **do**
  if $cluster.nbd_p > 0$ ***and*** $cluster.nbl_p \neq cluster.c$ **then**
    return $ERROR$;
  end
  if $cluster.nbd_n > 0$ ***and*** $cluster.nbl_n \neq max(N \times cluster.c, U)$ **then**
    return $ERROR$;
  end
end

user.

We used this algorithm on several Cloud configurations and compared it to other algorithms to see how it performs. Our Clouds configurations are the following:

- We set 1000 users per database.
- A database will need 4 cores to be installed
- We set the prices for processor and NUP licenses to 50 and 1 respectively to reduce big numbers but keeping the right magnitude order.
- We set the corefactor to 1 for each processor as it will not impact the simulation.
- We want to deploy up to 32 databases.
- Each Cloud is composed of n clusters containing m cores:

Algorithm 3: Step 3: Database cost optimization

input : Clusters: Array of clusters, U: Number of users
output: Clusters: Array of clusters
$Clusters \leftarrow SortedByTotalNumberOfCores(Clusters)$;
for ***all*** $Clusters$ ***as*** $cluster$ **do**
  $nbApps \leftarrow cluster.nbd_p + cluster.nbd_n$;
  if $(25 \times cluster.c > U$ ***and*** $nbApps > 2)$ ***or*** $(25 \times cluster.c < U$ ***and*** $nbApps > (50 \times cluster.c/U))$ **then**
    $ConvertAllToProcessor(cluster)$;
  else
    $ConvertAllToNUP(cluster)$;
  end
end

- $2 \leq n \leq 6$
- $32 \leq m \leq 256$ by step of 32

Different clusters have different m values, leading to $\sum_{i=2}^{6} \Gamma_8^i \approx 3000$ different Clouds.

We compared our algorithm to four others:

all random This algorithm takes a cluster at random and then randomly chooses the *metric* to use for the deployment.

cluster random This algorithm chooses at random a cluster of deployment but chooses the best *metric* for this cluster.

load balance This algorithm tries to put the same number of databases in all clusters.

smallest first This algorithm put the databases in the smallest cluster first (in term of number of cores) before moving to the next smallest one. Each time, it chooses the best *metric* to apply.

We can see in Figure 3.2 the results of this experiment. The SAM oriented algorithm is far better than the other and manages to pay only 9000 licenses in average whereas the second least expensive algorithm, $smallestfirst$, pays about 47,000 licenses. This shows the needs for SAM considerations in software deployment to optimize the software cost in companies. Here, with only one product, the SAM algorithm manages to reduce the expenses by about 80% while ensuring compliance at all times.

The conclusion to this experiment is that two choices exist: First, if we have few products to manage, it is extremely beneficial to develop an optimizing algorithm which takes all *metrics* into account and makes the best deployment possible in our Cloud architecture. On the other hand, if we have a lot of products to manage and the number of *metrics* is too large to have an efficient algorithm, we need another kind of deployment algorithm taking into account the *metric* model. We propose this kind of algorithm in the next section all the while introducing multi-parametric optimization.

3.2 *GreenSAM*: a multi-parametric deployment heuristic

In 2014, datacenters in the U.S. consumed an estimated 70 billion kWh, representing about 1.8% of total U.S. electricity consumption and datacenters in the world are responsible for 1% of all electricity. Current studies results show datacenters electricity consumption increased steadily by about 4% since 2010. Current trends estimate U.S. datacenters will consume approximately 73 billion kWh late 2020. As many IT services and tools use the Cloud and are dependent on large infrastructure that can be local or remote [36], more and more questions are being asked about the huge energy consumption of these infrastructures to supply and cool computing machine [37]. Research has been conducted to reduce these consumptions [38] and heuristics have been proposed such as

GreenPerf [39] for example which introduce a performance and power consumption ratio to improve energy efficiency or an energy-efficient framework dedicated to Cloud [40]. One other paper [41] proposed optimization of the placement of virtual machines with many other parameters including license costs and showed that handling both problems of mapping virtual machines to physical machines and mapping applications to virtual machines leads to better results than considering the two problems in isolation. Even so the problem is well formulated, it lacks modern SAM considerations and uses the fact that an application uses one license at most and that the number of licenses does not rely on the underlying architecture with is an unrealistic view of the modern SAM. Using only SAM considerations like in previous Section is not realistic for a deployment algorithm. We will, therefore, tackle this issue with a multi-parametric deployment algorithm which ensure compliance through all the Cloud environment by taking into account energy consumption and performance of applications.

As energy consumption and product performance are a completely separated academic domain, we will use variables for energy and performances. The goal is to propose an algorithm taking into account these parameters without computing them. The fi-

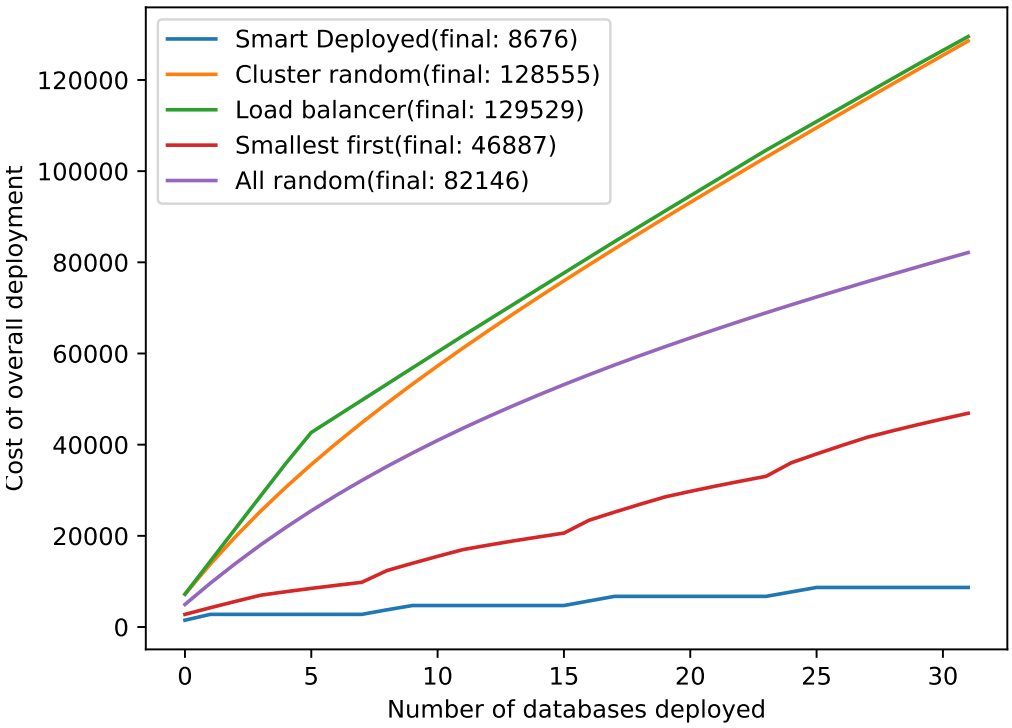


Figure 3.2: Comparison of algorithms for the deployment of 32 Oracle Database on about 3000 different Clouds

nal user will be able to use state-of-the-art estimation algorithms to fill the variables. Therefore, we consider that we are responsible for the energy consumption of all servers we put the application on. The first application will consume all the energy of the server and the following applications installed on the same server will not consume anymore energy. The performance variable is defined with *Performance Indices (PI)*. We categorize products in several classes and each class has a performance computation algorithm. For example, a Cloud storage product will require a lot of storage and bandwidth while a database could require a lot of CPUs resources. We define *PI* for each product we want to deploy to better match our performance needs. A product is, therefore, defined by its *metrics* and its *PI*. The behaviour of energy consumption and *Performance Indice* are showed in Figure 3.3 and Figure 3.4 respectively.

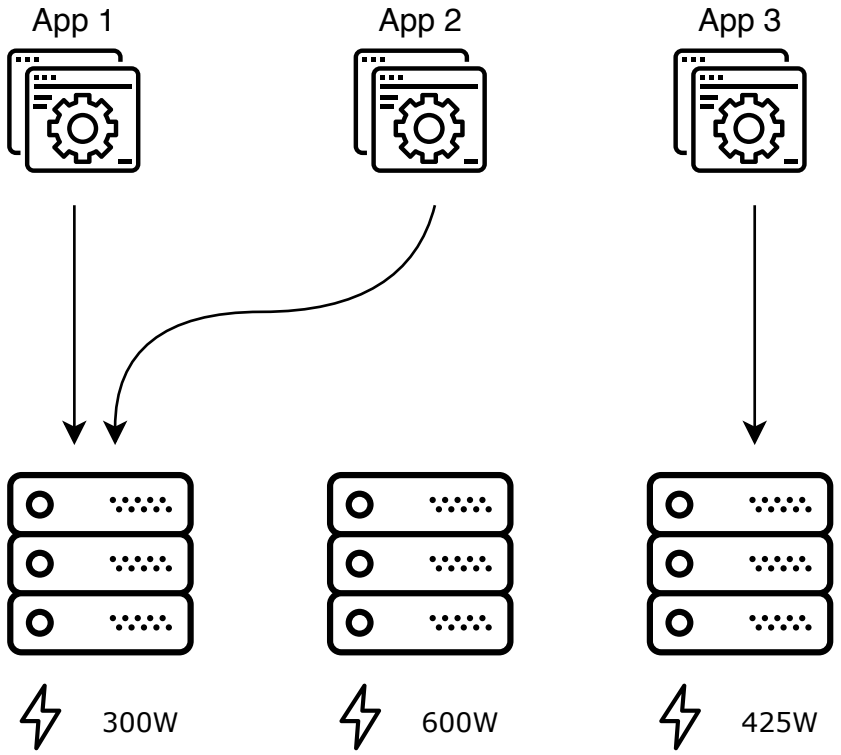


Figure 3.3: Example of energy consumption computation: Here, *App1* will consume 300W when its deployed while *App2* will not consume anything. As there is no more space on the first server, *App3* have to go on another server, consuming 425W.

Any optimization problem will have design parameters whose best possible values from the viewpoint of the objectives are sought to be attained in the optimization process. The optimization task here is to map a set of software onto available resources, here servers. The three objective functions are defined with the following variables:

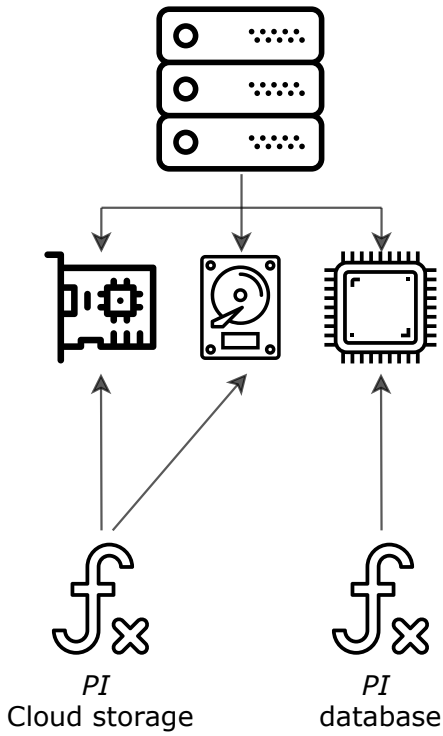


Figure 3.4: *Performance Indices* of software: Here a database PI is a function that takes the number of cores in argument while a Cloud storage PI requires storage and network attributes

.

n The number of servers noted s.

m The number of applications noted a.

A_s The attributes of a server s.

f_a The formula for the *metric* of the application a that takes into account attributes of servers.

E_s The energy consumed by the server s for the first installation. As stated before, when we deploy and application a on a server already containing one then $E_s = 0$.

PI_a The performance indice of the application a. It is a function waiting for attributes of a server to give performance score. If the application a is not installed on the server s then $PI_a(A_s) = 0$. Besides, to avoid putting all applications on the same server, another operational constraint states that an application requires 2 cores not already used. If the application a doesn't fit on the server s then $PI_a(A_s) = 0$. This constraint can be modified to fit any needs.

E_T The total energy consumed by the deployment.

P_T The total performance score of the deployment.

L_T The total number of license needed by the deployment.

The first objective function is the minimization of energy consumption: the total energy consumption 'E_T' of our deployment is expressed as:

$$E_T = \sum_{s=1}^{n} E_s \times (\neg (\exists a \in [1..m]/a \in s)) \tag{3.12}$$

The second objective function is the maximization of performance. The overall performance P_T is expressed as:

$$P_T = \sum_{s=1}^{n} \sum_{a=1}^{m} PI_a(A_s) \tag{3.13}$$

The last objective function is the minimization of software cost: this objective will stop the process if the *metric* computation brings a noncompliance state warning. The total license consumption L_T is expressed as:

$$L_T = \sum_{s=1}^{n} \sum_{a=1}^{m} f_a(A_s) \tag{3.14}$$

In most cases, machines that bring performance will have higher energy consumption, implying that objectives P_T and E_T are contradictory - forming the basis for multi-objective optimization. The variable $f_a(A_s)$ is the computation of the *metric* model on a specific server. Each time we want to deploy a product, we compute the three criteria for each server before using a Pareto front to keep a subset of best servers. We have to compute the three criteria each time we deploy a new product because the variables are likely to change like the performance that will decrease if we put a lot of products on the same server. From the subset of servers, we filter the ones that will put us in a noncompliant situation. If the subset become empty after this step, it means that the product we want to deploy will always create a noncompliant state meaning we have to skip it and warn the user.

We still have to sort the remaining servers, so we implemented the *GreenSAM* heurisitc which will give from the three criteria a score to sort the servers. This score function will divide the normalized performance of the server by the sum of the normalized energy consumption and the normalized license consumption:

$$Score = \frac{\frac{P_s}{M_P}}{\frac{E_s}{M_E} + \frac{L_S}{M_L} + 1} \tag{3.15}$$

with the following variables:

E_s Energy consumption of server s.

L_s License consumption of server s.

M_E Maximum of energy across the server subset.

M_L Maximum of license consumption over the server subset.

M_P Maximum of performance over the server subset.

P_s Performance of server s.

Finally, *GreenSAM* will take the first server of the resulting sorted array and will place the product on it. For example, given the set of servers defined in Table 3.1, we obtained the Pareto front represented in Figure 3.5 and, therefore, the subset is composed of servers 2, 4, 7 and 8.

Table 3.1: Set of servers

id	$performance$	$energy$	$licenses$
1	8	300	16
2	16	300	16
3	4	550	12
4	32	375	12
5	4	550	16
6	8	375	16
7	4	300	8
8	8	400	4
9	8	325	16
10	32	475	16

With this subset we can now compute the scores for each server giving us the result in Table 3.2 so the product will be deployed on the server 4 before handling the next one.

Table 3.2: Set of servers

id	$performance$	$energy$	$licenses$	$Score$
2	16	300	16	0.182
4	32	375	12	0.372
7	4	300	8	0.056
8	8	400	4	0.111

Note that as we do local optimization, we will not obtain the optimal deployment of all products by taking them one by one. If two servers get the same score, we will randomly choose one but it means that the user should change the PI calculation to get

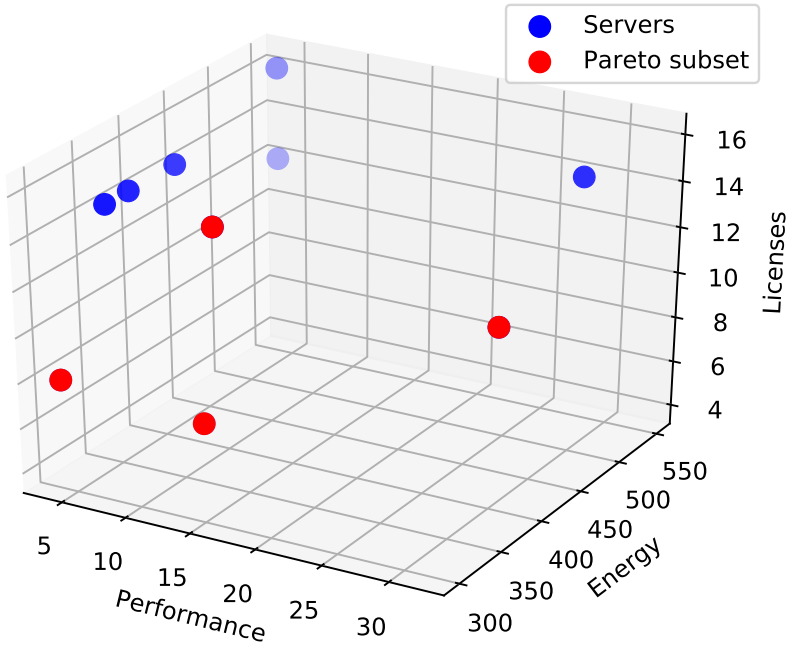


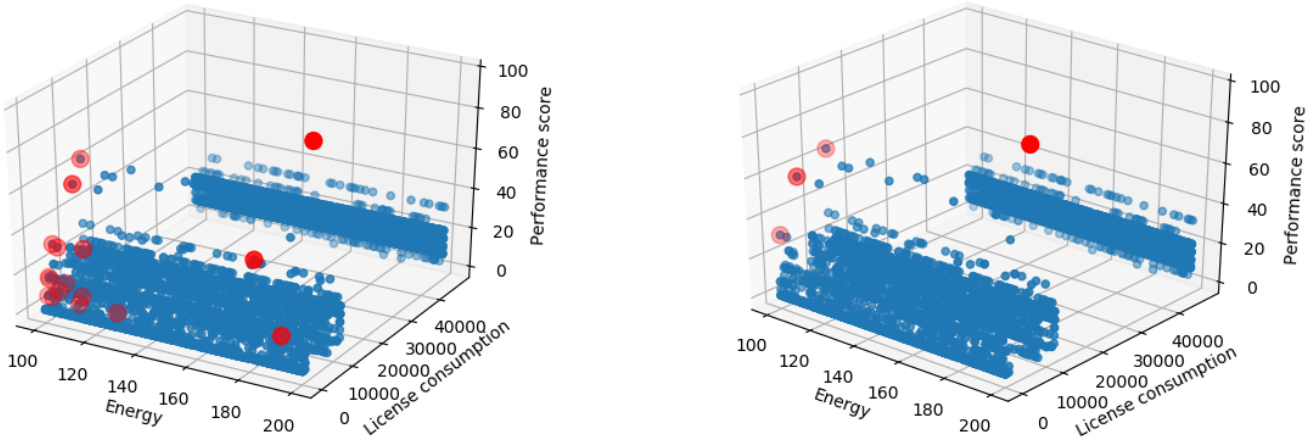
Figure 3.5: Pareto front on the server set defined in Table 3.1

a finer result or add a parameter to differentiate the servers which have the same scores. We evaluated this heuristic on two uses cases: Oracle Database Enterprise Edition as we used its *metrics* in this document for the examples and RedHat OpenStack to see the impact of using products with different *metrics*.

3.2.1 Oracle Database Enterprise Edition use case

For both use cases, we used a server data set coming from the french Orange™ company (first historical French multinational telecommunications corporation) with more than 5000 servers. In this use case we will deploy 10 databases in the Cloud and compare the *GreenSAM* heuristic to others. The *Performance Indice* for the database will be the following: we must have a minimum of 2 cores to be eligible and the performance score is the number of cores divided by the number of already installed databases on it. The heuristics we compare *GreenSAM* to are the following:

PerfEnergy This algorithm promotes performance first, the, energy meaning that we will choose the most performant server and if two have the same performance, the one which consume the less energy.



(a) First deployment

(b) Fifth deployment

Figure 3.6: Pareto front of Oracle Database deployment

LicPerf This one focuses on the license consumption before optimizing the performance.

RatLic This algorithm optimize the ratio between performance and energy before optimizing license consumption.

We can see in Figure 3.6a the Pareto front of the first Oracle Database deployment in the Cloud. Only 17 servers (red circles in Figure 3.6a) were part of the subset. This number went down to 4 (red circles in Figure 3.6b) for the fifth deployment as shown in Figure 3.6b.

The final results of this deployment is presented in Figure 3.7. We see that in the end, *GreenSAM* loses half the performance compared to the best algorithm performance wise. In return, it manages to lower the energy consumption to a quarter of the energy consumed by the most energy-intensive algorithm. Besides, it gets good results in the license consumption criteria. Overall, in this example, *GreenSAM* sacrifices performance to enhance its energy and license consumption. In the next example, we will see a more realistic deployment of multiple products that forms a usable environment.

3.2.2 RedHat OpenStack use case

In this use case, we will deploy a minimal OpenStack platform with the *metrics* from the RedHat editor. The purpose of this deployment is to have a lead node (called director), ten compute nodes and twenty CEPH nodes. Each node has its own *metric* and $PerformanceIndice$ defined as follows:

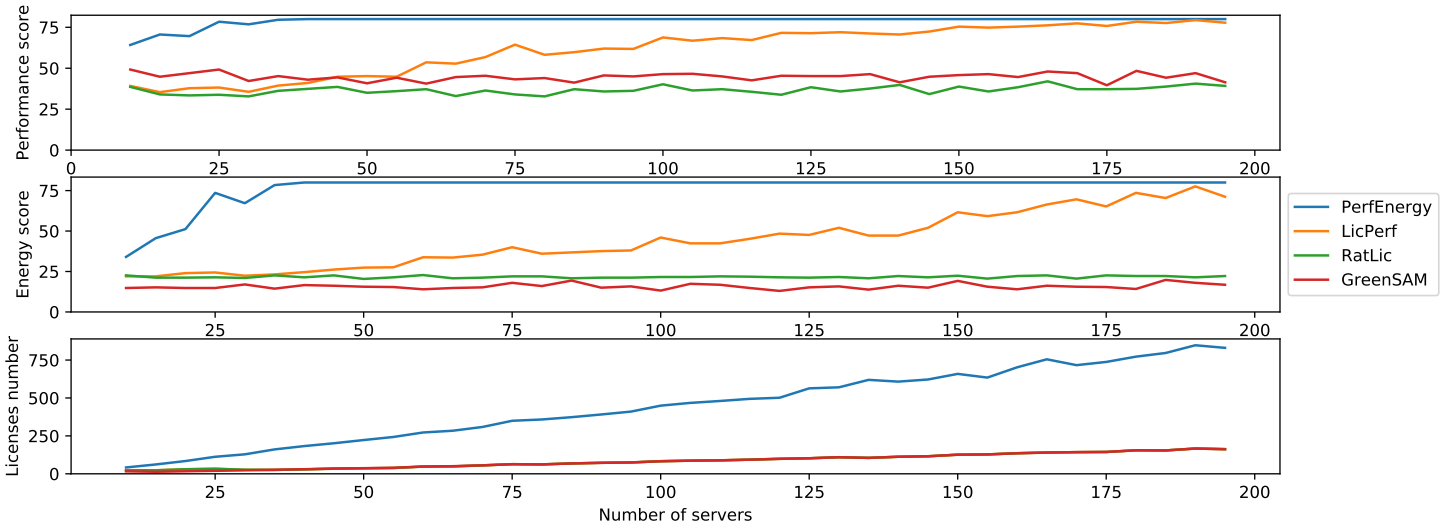


Figure 3.7: Result of Oracle Database deployment using *GreenSAM*

Director node It must be located in a cluster with as many servers as possible to be able to deploy as many compute and CEPH nodes as desired. Therefore, the number of servers in the parent cluster will be the score.

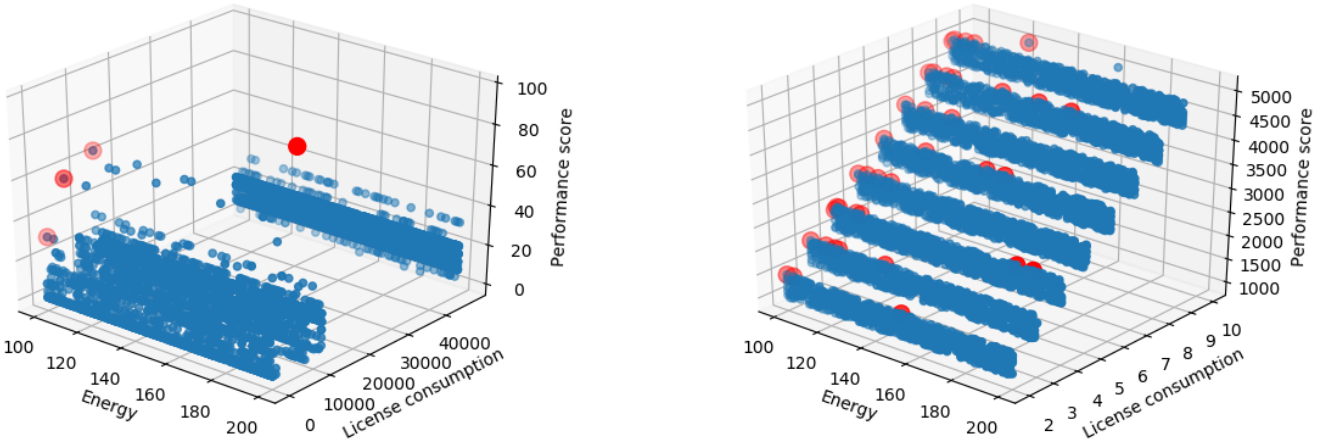
Compute node It needs a lot of cores and must be deployed on a server without anything else on it. The score will be the number of cores.

CEPH node It requires a lot of storage. The score will be, therefore, the amount of storage available. As the compute node, we cannot reuse a server twice.

The licenses are computed as follow: we need one license per director node and compute node while the CEPH nodes requires 1 license per 500 memory slots. We can see in Figure 3.8a and Figure 3.8b the Pareto front for the first deployment of the compute node and CEPH node respectively.

Finally in Figure 3.9, we can see the good results of the *GreenSAM* heuristic. In this use case also, *GreenSAM* manages to have acceptable performance while reducing energy consumption and licenses. Besides, compared to the other algorithms, *GreenSAM* makes a good deployment while ensuring compliance at all time which is not the case of the other algorithms.

We demonstrated in this section that a SAM oriented deployment heuristic, *GreenSAM*, can optimize multiple parameters while ensuring compliance. Even with local optimizations, this heuristic is better than others and is usable in real case scenario with Clouds of over 5000 servers. We will enhance this deployment algorithm in the next section by deploying multiple product in one time to try to reach the global optimal deployment.



(a) Compute node (b) CEPH node

Figure 3.8: Pareto fronts for RedHat OpenStack deployment

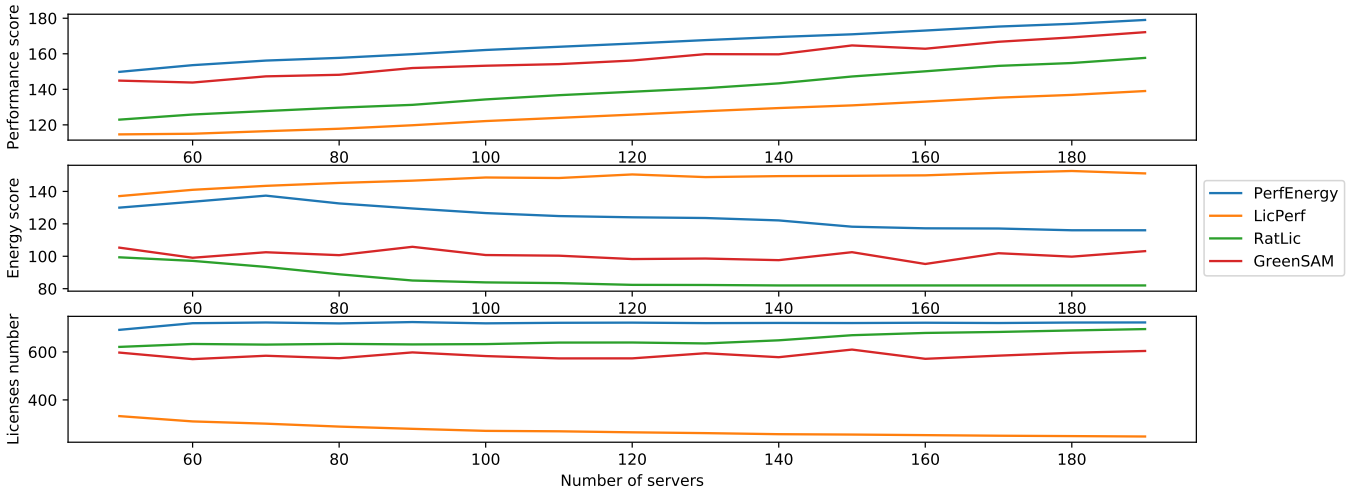


Figure 3.9: Result of RedHat Openstack deployment using *GreenSAM*

3.3 Deploying multiple products at the same time

We will start by proving that deploying multiple products at one time while optimizing the three above criteria in a Cloud is a NP-complete problem. In order to prove that, we define a new problem with the following variables:

n Number of products to deploy

s Number of servers

α_i Cost of license for product i

β_j Cost of using one core on server j, energetically speaking

$m_{i,j}$ Number of licenses consumed for product i on server j. If product i is not on j tthen $m_{i,j} = 0$

r_i Number of cores required by product i

R_j Number of cores available on server j

Definition 1. *The optimization problem SAMDeployment is defined by the following equations:*

$$min \sum_{i=1}^{n} \left[\left(\sum_{j=1}^{s} m_{i,j} \right) \times \alpha_i \right] \tag{3.16}$$

$$min \sum_{server\ j\ used} R_j \times \beta_j \tag{3.17}$$

subject to:

$$\forall\ j \sum_{all\ deployments\ i\ on\ server\ j} r_i \ \leq \ R_j \tag{3.18}$$

which is a physical constraint where the cores used from deployment on a single server cannot exceed the number of cores of that server.

and the associated decision problem:

Definition 2. *The decision problem SAMDec is defined as: given two constraints, B_E and B_L which are energy budget and license budget respectively, is there a deployment that fulfill the following equations:*

$$\sum_{i=1}^{n} \left[\left(\sum_{j=1}^{s} m_{i,j} \right) \times \alpha_i \right] \leq B_L \tag{3.19}$$

$$\sum_{server\ j\ used} R_j \times \beta_j \leq B_E \tag{3.20}$$

also subject to Equation.3.18.

We then declare that this decision problem is NP-complete and prove it:

Lemma 2. *SAMDec is NP-complete.*

Proof.

Definition 3. *Let I_1 be an arbitrary instance of $2PARTITION - EQUAL$ with:*
$2n$ integers $a_1, ..., a_{2n} \leq 1$ where $\sum_{i=1}^{2n} a_i = 2S$

$$\exists\ I \text{ subset of } \{1, ..., 2n\} \ /\ |I| = n \quad \sum_{i\ \in\ I} a_i = S$$

Definition 4. *Let I_2 be an instance of $SAMDec$ with:*
n products to deploy with $r_i = 1 = m_{i,j}$ and $2n$ servers with $R_j = 1$ hence one product per server and $s = 2n$.
With $Used$ declared as the set of used servers (by indices), $n = |Used|$ iff there is a solution:
Let $\beta_j = a_j$:

$$\sum_{server\ j\ used} R_j \times \beta_j = \sum_{server\ j\ used} a_j = S$$

Let $\alpha_i = X - a_i$:

$$\sum_{server\ j\ used} \left[\left(\sum_{software\ i\ on\ j} m_{i,j} \right) \times \alpha_i \right] = n \times (X - a_i) = nX - S$$

Moreover, if I_2 has a solution with B_E and B_L fixed then we have a solution $Used$ for the following equations:

$$\sum_{j\ \in\ Used} \beta_j \ \leq\ B_E$$
$$\sum_{j\ \in\ Used} \alpha_j \ \leq\ B_L$$

With the following:

- $Size(I_1) = 2n + log\ \sum_{i=1}^{2n} a_i$
- $Size(I_2) = 2n + log\ \sum_{i=1}^{2n} \alpha_i + log\ \sum_{i=1}^{2n} \beta_i$

We can conclude that I_2 has a polynomial-size in I_1 and we prove that I_1 has a solution $\iff$ I_2 has a solution:

I_1 has a solution I

$|I| = n$ and $\sum_{i \in I} a_i = S$

We take $Used = I$ and have a solution to I_2

because:

$$\sum_{j \in I} \beta_j \times R_j = \sum_{j \in I} \beta_j = S$$

$$\sum_{j \in I} \left[\left(\sum_{metric\ i\ on\ j} m_{i,j} \right) \times \alpha_i \right] = n \times \alpha_i$$

$$= nX - S$$

hence a solution to I_2.

I_2 has a solution $Used$

$|Used| = n$

We take I as $Used$ and have a solution to I_1 because:

$$\sum_{j \in Used} \beta_j = \sum_{j\ in\ Used} a_j = S$$

$$\sum_{j \in Used} \alpha_j = nX - \sum_{j \in Used} \beta_j = nX - S$$

altogether with $|Used| = n$, we have a solution to I_1.

□

This proves that the problem of deploying one product per server and only optimizing energy consumption and license cost is NP-complete. As our problem adds the performance criteria and allows multiple products per server, it is a sur-class of the *SAMDeployment* problem and, therefore, is NP-complete too.

We tackle this NP-completeness by proposing an alternative model of multiple products deployment in Cloud environment coupled to the *GreenSAM* heuristic. This tree-shaped model represents all the possibilities of deployment and handles all our constraints. Each layer of the tree is the deployment of a product and each node is an available server to deploy on. The root node is the starting point and the edge between two nodes represents the deployment of a product on a server. Each node has attributes which are the criteria, performance, energy, and license. We handle the dynamicity of criteria by modifying the attributes on different layers: Sons will have their attributes modified according to the parent server, meaning that if we take a node representing the server s_1 and follow the edge to a node representing the same server, then the son will have its performance attribute modified. As the deployment of a product on a server

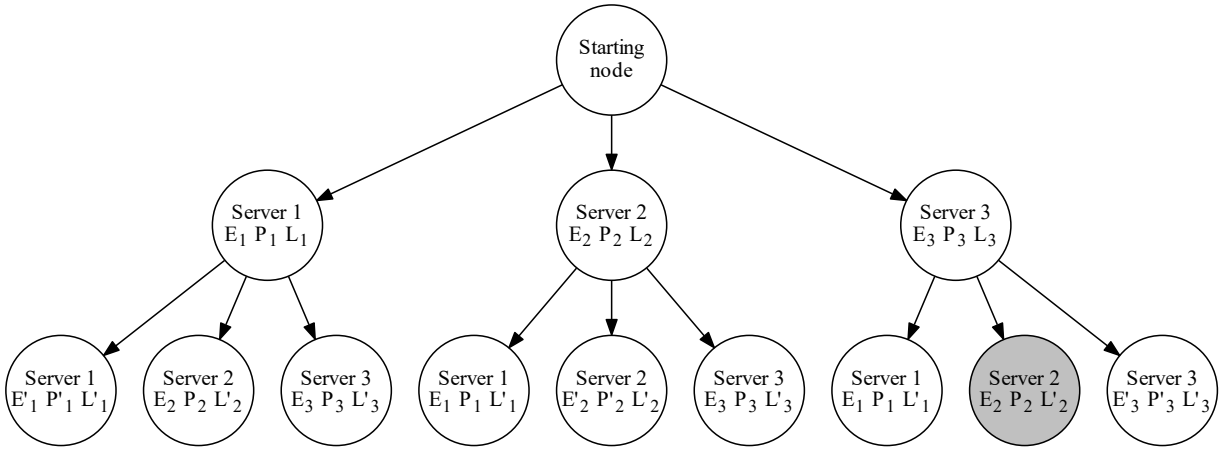


Figure 3.10: Example of deployment model with three servers and 2 products to deploy: We can see that if we are in the grey node, it means that we deployed the first product on server 3 and the second product on server 2. Therefore, our final criteria scores will be: $E_T = E_3 + E_2$, $P_T = P_3 + P_2$ and $L_T = L_3 + L_2$.

can impact other server, the license consumption of all servers have to be computed for each layer. The Figure 3.10 clearly shows this model with the variations of attributes.

While this model allows to easily represents the deployment of multiple products on a Cloud, it uses also a lot of space. Figure 3.11 describes the space usage depending on the number of servers and number of products to deploy. We can see that the memory usage of the tree explodes quickly with tens of GB for 150 servers and 4 products.

As it is not realistic to go through the entire tree to search for optimal value, we have to find a heuristic to find this optimal quickly. We will use the *GreenSAM* heuristic described in the previous Section and explore the nodes in the order given by it. Each time we explore a node, we use the heuristic to sort the sons and go through them in order until reaching a leaf. If the score of the leaf is better than the overall found score then we store it and continue the search until two possible limits: Either a time limit where we let the algorithm search for a good value for a specific time, or a memory size limit where we allow the algorithm to go through a specific number of nodes. This traversal algorithm is described in Algorithm 4.

With the sheer number of 2,227.33 PB of memory needed for a deployment of 10 products over 50 servers, we evaluated this deployment algorithm to three other algorithms:

Bruteforce This algorithm go through every node in order from left to right. This solution can be very efficient (if the optimal is right at the beginning) or very inefficient (in the other case).

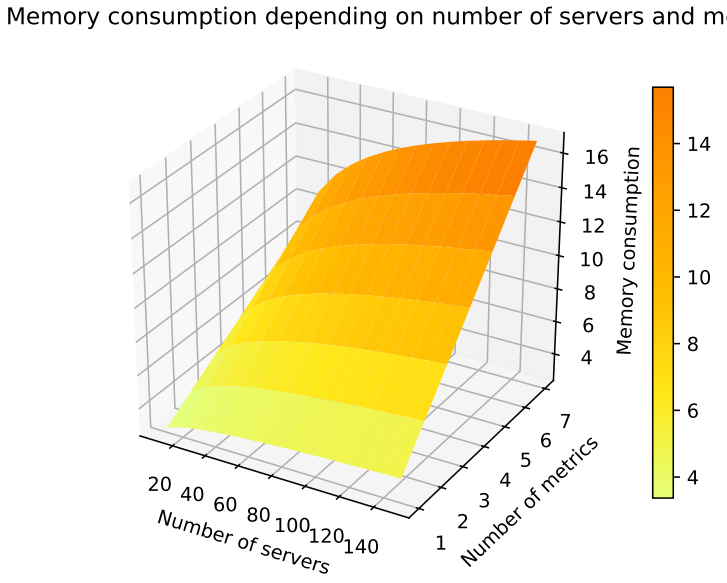


Figure 3.11: Memory consumption of the tree-shaped model expressed in log10 of bytes

Algorithm 4: Tree model traversal algorithm

```
Function TreeTraversal(Node n, Global FinalResult) : int is
    if  return FinalResult ;
     then either limit reached
    end
    if n has no son then
        FinalResult ← max(n.score, FinalResult) ;
        return FinalResult ;
    end
    subset ← ParetoFront(Sonsofn) ;
    for Son s in subset do
        s.score ← GreenSAM(s, subset) ;
    end
    subset ← sort subset by descending score ;
    for Son s in subset do
        TreeTraversal(s) ;
    end
end
```

MinLicenses This algorithm go through the nodes that consumes the least amount of

licenses first.

Pareto This algorithm uses a Pareto front to create a subset of servers before bruteforce its way down the tree. Each layer it creates the subset then go through it in order from left to right.

To be more precise about memory use and scoring, we developed a simulator which builds the tree and launch algorithms against it. It can be configured with a specific seed to get the same data set over and over and so give reproducible results. This simulator takes 44 bytes of memory by node, which explains above memory consumptions. This could potentially be optimized but will not impact the simulation as we always compare heuristics on the same ground. We will use the score function from Equation 3.15 to establish what is a 'good' result but compares algorithms by criteria to see how they behave. Besides, because of the format of the scoring function we can deduce if we are close or not from the optimal. Inded:

$$Score = \frac{\frac{P_s}{M_P}}{\frac{E_s}{M_E} + \frac{L_S}{M_L} + 1}$$

returns a result in the range $[0; 1]$ and if we have the most performance server (so $P_s/M_P = 1$) and both license consumption and energy consumption are 0 then the result is a perfect 1. When the result tends to zero, it means that the server we want to deploy on is terrible. Using that, with a deployment of n servers, the score of deployment will be in the range $[0; n]$. Most of the time, the optimal solution will be inferior to n as we don't have, in real-world situations, servers that will not consume anything during the deployment of all products. But we know that the closer we get to n, the closer we get to the optimal. All evaluations below used the simulator and compute the average of 20 different sets per point.

The two first experiments used the memory limit where we specified a maximum number of nodes to go through and compared algorithms. The first one compared them with a data set where we could compute the optimal solution to see how far we are from it. In Figure 3.12, we can see the results of the deployment of 5 products over 20 servers with a variable amount of memory allowed.

We then stressed the algorithms by making a huge deployment of 50 products over 1000 servers at one time and compared the results. We fixed very low memory amounts from 1MB to about 300MB compared to the memory needed to fit the tree which reached 2e+152 bytes. We can see the results in logarithmic scale in Figure 3.13.

The third and fourth evaluations used time limits to see how quickly the algorithms get good results. Like the first experiment, we started by comparing results to optimal. We fixed time limits from 500 milliseconds to 3 seconds for the deployment of 5 products over 50 servers and presented the results in Figure 3.14. The last evaluation uses the same big sets of the second experiment and the same time limits. We can see the results in Figure 3.15.

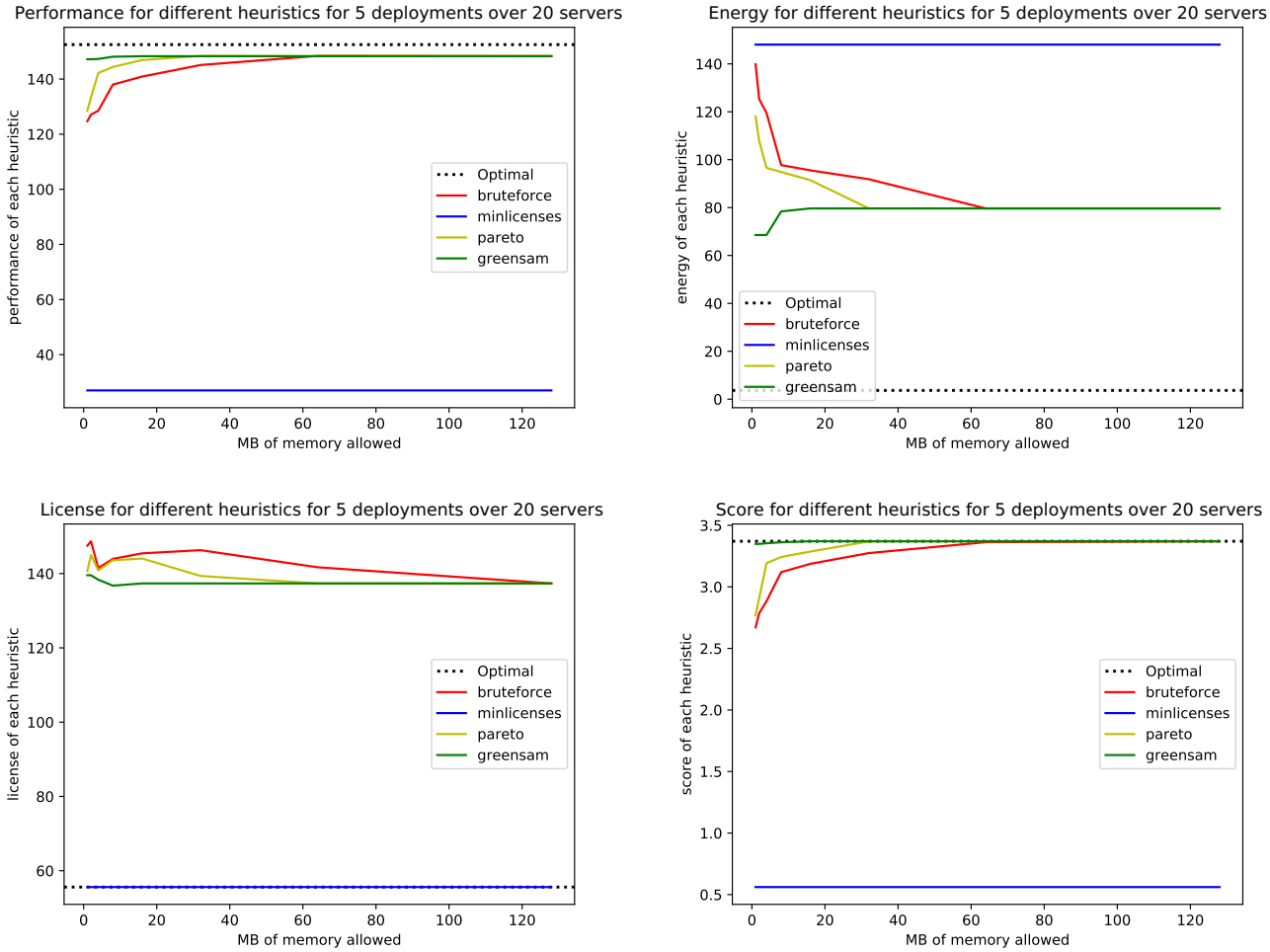


Figure 3.12: Deployment of 5 products over 20 servers with a memory limit: We can see that overall, the *GreenSAM* heuristic obtains the best results on all criteria except in licenses where obviously, the *MinLicenses* algorithm is better. While the others algorithms take times to stabilize, *GreenSAM* find quickly its best solution and is very close to the optimal performance.

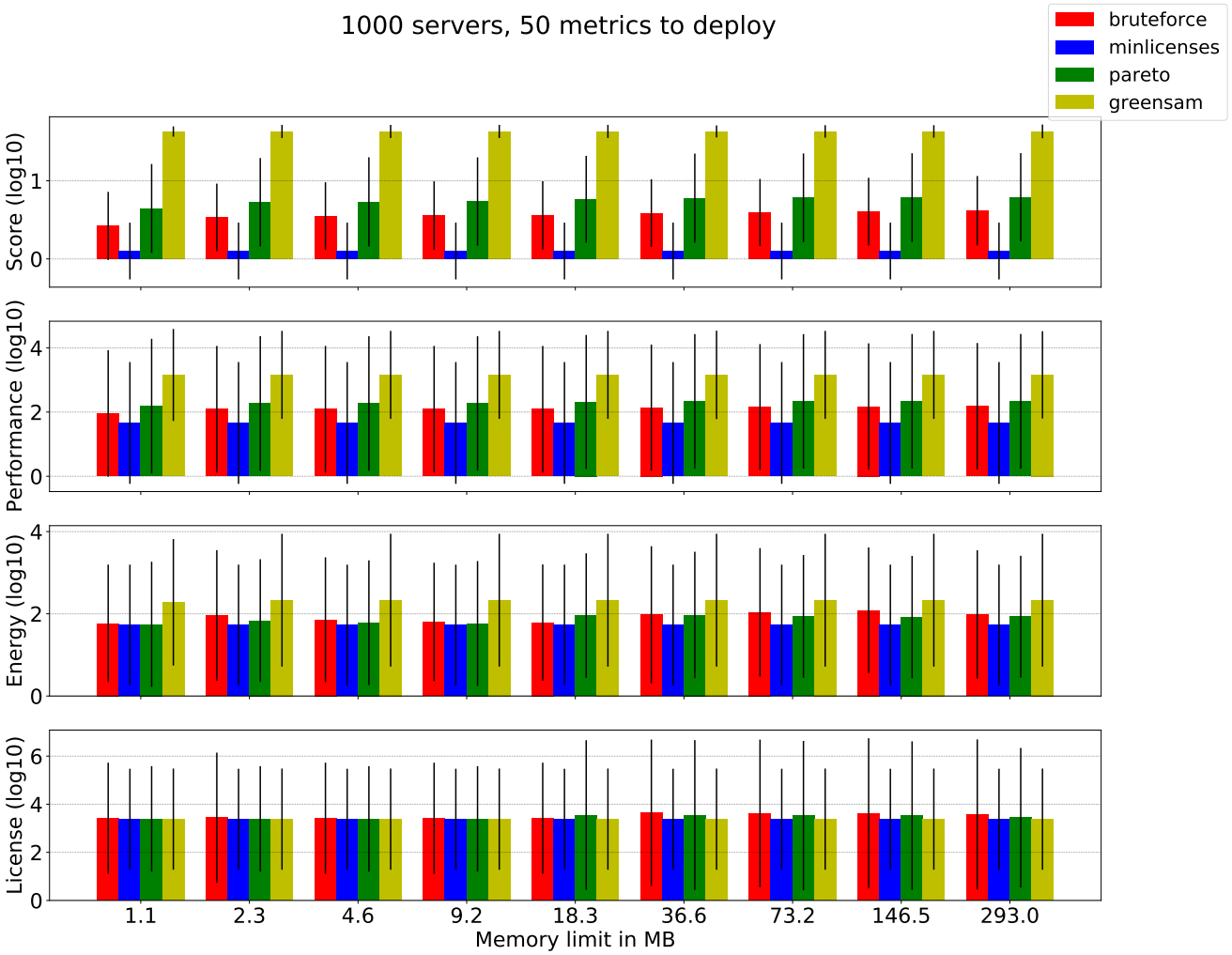


Figure 3.13: Deployment of 50 products over 1000 servers with memory limit: The final score of the *GreenSAM* heuristic was 43.23 while the others algorithms got 2.6, 0.13 and 10.7. With just 1MB of memory, meaning going through 57,000 nodes over 50^{1000}, *GreenSAM* finds a very good solution that has the best performance overall and very few licenses at the cost of energy.

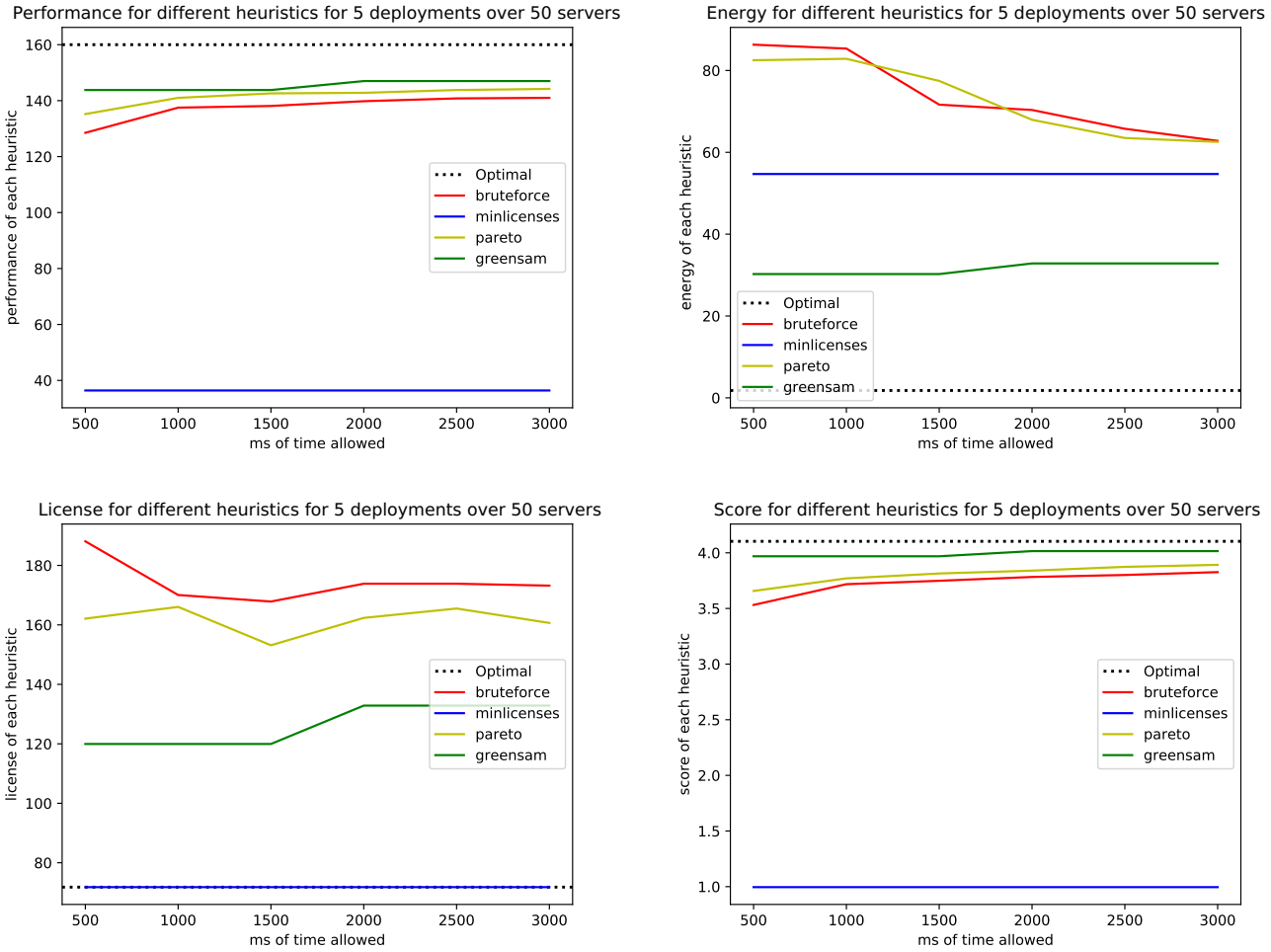


Figure 3.14: Deployment of 5 over 50 servers with time limits: We see that *GreenSAM* beats the other algorithms in this configuration too. It stabilizes less quickly than with memory limit but manages to get very good score, performance, and energy while sacrificing few licenses.

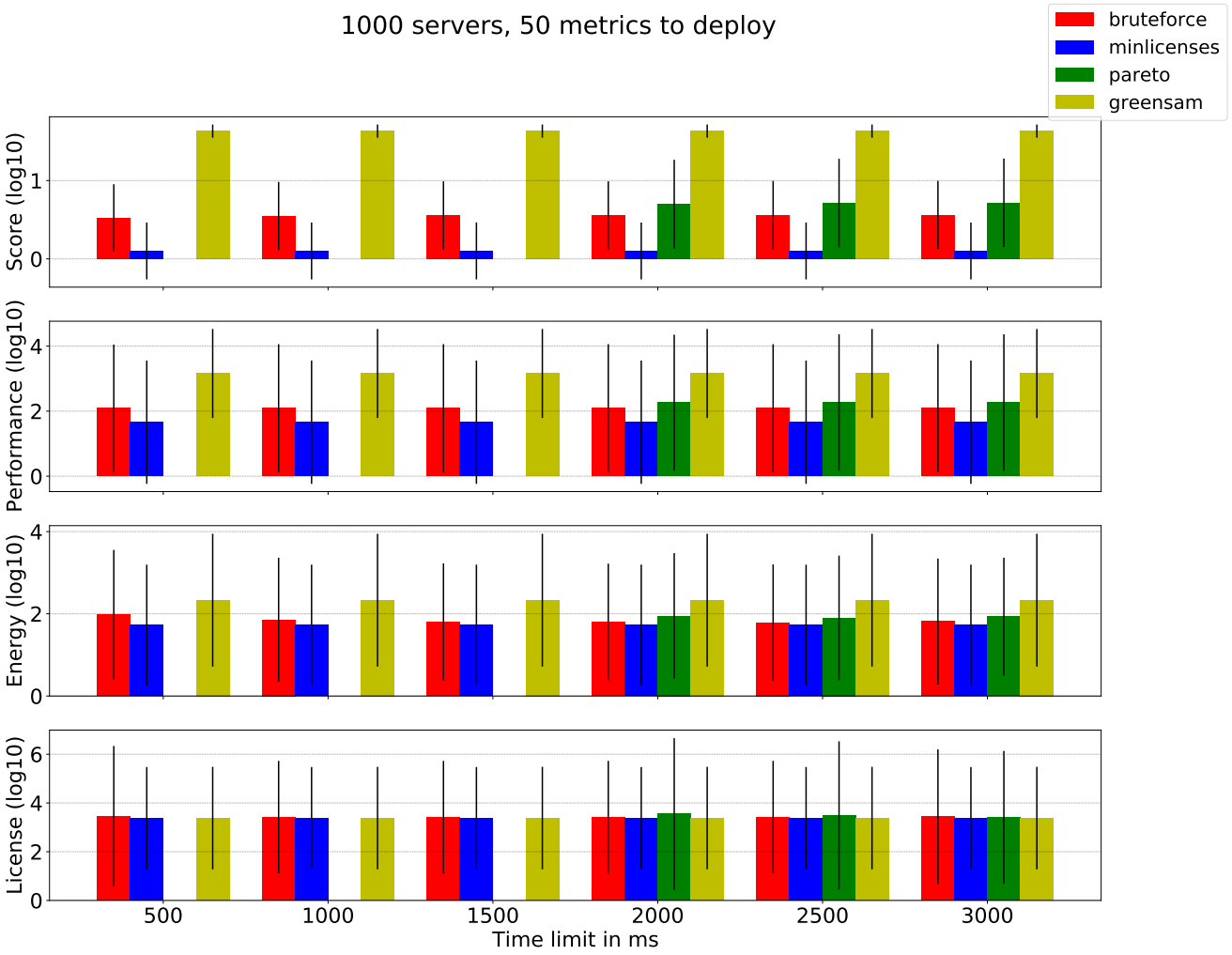


Figure 3.15: Deployment of 50 products over 1000 servers with time limits: We can see that under 1.5 seconds the *Pareto* algorithm don't manage to reach a single leaf because of the amount of computation to do but once it does, it gets good results. The *GreenSAM* heuristic is also the best overall in this configuration and reaches the score of 43.17 with only 500 milliseconds.

3.4 Forecasting

Following these results, we tried to answer the other part of the problem: what happens when the Cloud changes, either by adding, removing or modifying resources after deploying products. Even if our algorithms manage to make a very good deployment, they are based on a snapshot of the Cloud state. If it changes, it can have serious impact on the required number of licenses and, therefore, on the compliance. These considerations express the need to have a forecasting tool warning us if, based on the Cloud modification, we will soon be in a non-compliant state. This can happen when the required number of licenses go over our threshold (our license stock). In this section, we present our work to address this issue.

We tackled this problem with an intern, Irfaane OUSSENY, and compared several algorithms to see how they compared on license consumption curves. Each algorithm will warn us when a threshold is about to be exceeded by putting a flag on the time axis. Each flag in Figure 3.16 represent a different situation. We can see that we are going over the threshold at the time Td. We will use this time to represent good flags and bad flags. Obviously, the flag $F3$, arriving after the exceedance, is a bad flag. Another bad flag is $F1$ because it arrives too soon and is considered as a false warning as opposed to $F2$ which is a good flag. The difference between $F1$ and $F2$ is expressed with the variable x representing the time limit that separates a false warning from a good flag. A flag between $Td - x$ and Td is considered a good one.

We used this representation to give a score to flags given by the different algorithms

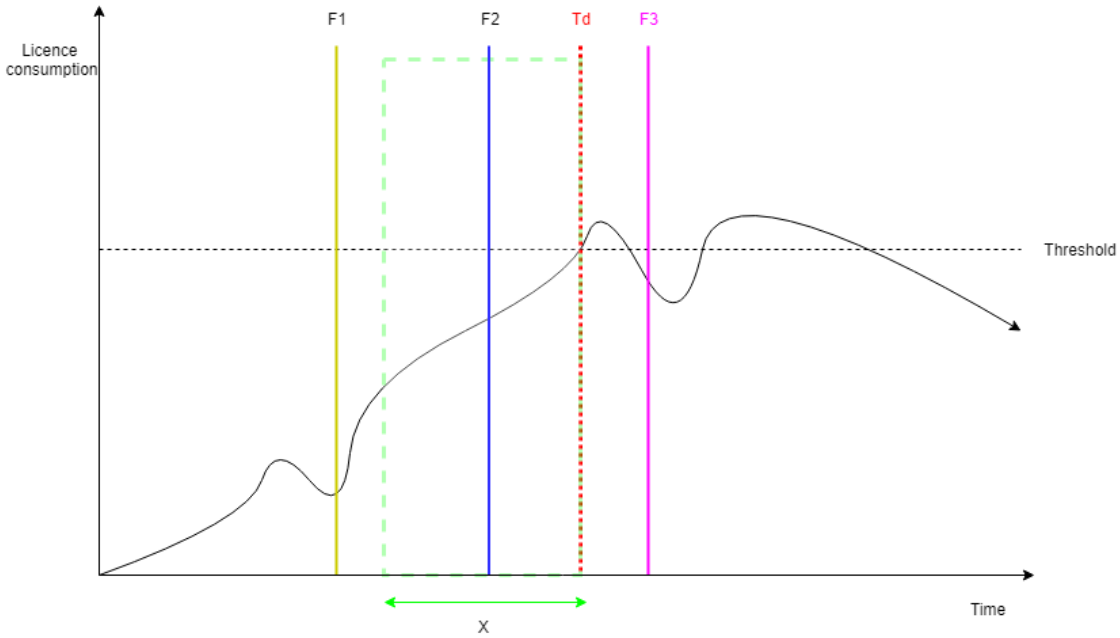


Figure 3.16: Different flags declared by the forecasting algorithms

to test them. Each flag will have a score depending on its position compared to Td. Given a penalty P and a time limit x, this score function is expressed as:

$$S = \begin{cases} Td - f, & \text{if } Td - x \leq f < Td \\ -\infty, & \text{if } f \geq Td \\ -P, & \text{if } f < Td - x \end{cases} \tag{3.21}$$

This equation allows to severly punish a late flag and gives a penalty to a false alarm. The score of a good flag is computed with its position inside the time limit. The farther it is from Td, the more points he gains. We compared three algorithms: a naive one, the linear regression and a polynomial regression. The naïve algorithm compute the average gradient between the last x points. It then computes if the curve, given this average gradient, will get above the threshold soon. The polynomial regression was tested with various degrees and we took the best results each time. We then compared these three algorithms on thirty different license consumption curves. We can see in Figure 3.17 that the scores are very different from one curve to another. This can be explained by the fact that licence consumption is very dynamic and subject to many variations caused by unpredictable actors. Because these simple forecasting techniques produce highly

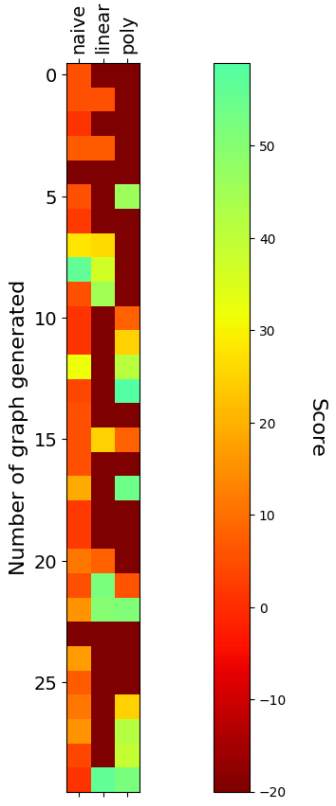


Figure 3.17: Results of the forecasting algorithms

variable results, and given the magnitude of the problem, it is necessary to look for new ways to forecast stock overruns.

In conclusion, we started by proving that creating SAM oriented deployment algorithms can cut software expenses in companies before presenting a deployment algorithm that ensure compliance while optimizing these license costs savings. We then introduced a heuristic that optimizes several criteria and manages to get good results with the deployment of one product on the Cloud compared to other heuristics. This heuristic, *GreenSAM*, also ensures compliance during the deployment. Following these results, we addressed the problem of deploying multiple products at once while optimizing the same criteria and, obviously, keeping the compliance. We proved that the decision problem of such deployment is NP-complete and proposed an algorithm getting very good results with little time and low memory limit. Finally, we introduced the forecasting problem by stating it and showing that simple forecasting algorithms gave very different results. This showed, given the stakes with compliance, the need of finding new ways to predict stock overrun accurately. In the next chapter, we will present how our deployment algorithm can be used effectively in the Cloud thanks to a generic SAM oriented tools optimized for Cloud architectures.

Chapter 4

Software Asset Management tools

Contents

Several Software Asset Management (SAM) tools editors exist on the market. Those editors have products that will manage our software park and give us reports about our compliance like SmartTrack from Aspera [42], Snow License Manager from Snow Software [43] and FlexNet Manager from Flexera [44]. Each tools have strong and weak points, for example Snow and Flexera both have an integrated discovery and inventory tool. Flexera, following the acquisition of a security firm, can now warn us about security vulnerabilities on the products we use. Snow Software have the Software Recognition Service (SRS) which is able to identify products accurately thanks to a huge database containing almost 600,000 products signatures from around 98,000 software publishers. In 2017, Vion [22] made a review of existing tools that showed that this editors were not ready for the Cloud. The best tool in the market could barely manage 10,000 devices at the same time, including servers, phones and laptops. With the rise of Software Defined Network and Virtual Network Functions, this number was not enough for big companies and, therefore, the need for a new Cloud intensive oriented tool appeared. Three years later, all the major editors cited before updated their tools to manage Cloud and optimization of licenses. Based on the result from [45], we introduced the tool called OpTISAM.

4.1 Motivation

Monitoring the use of our software is very complex when it comes to the Cloud. The choice of how to follow this usage is a vital step in SAM. The definition of this usage and the scope of our SAM actions are of equal importance because if we lack data at the time of contracting, it may be too late to go back. As stated before, traditional license management mechanisms typically do not match with essential Cloud characteristics

such as virtualization, elasticity, self-service and on-demand consumption. Virtualization brings technical challenges to map physical licenses models to a virtual environment and adapt the cost models. The on-demand resource allocation in the Cloud complicates any prediction of software licensing costs. Besides, probability to breach entitlements based on such usage or equipment characteristics, exponentially increase.

Then comes the problem of data storage. How long is the data storage of the users data? Given the possible size of a Cloud and its rapid expansion, this question becomes critical. These data are obviously very sensitive and must be accessible for a few years without deterioration. These problems are not an issue for Cloud storage experts but this is an additional cost we have to plan to use our software in compliance with our license agreements and it becomes very problematic when we didn't think about it. This was still possible because of the speed of the audit process. It can take up to months for editors to check the compliance of major companies but with new algorithms and technologies, this computation time just dropped to seconds or minutes.

The last issue is the data problem. An effective SAM will need quality and accessible data for many aspects ranging from simple compliance verification to computations to optimize our use. This require an aggregation of a lot of data sources (physical and logical inventories, contracts, project management, etc.) and will create entanglement if not done correctly.

Theoretically, the data problem is more complex. How do we know when a data will be useful, should we store it to manage possible future contracts? As such, we cannot filter any data during the aggregation phase because it could impact a future contractualization if we do not have the data for the computation of the *metric*. As a result, the database must be able to collect a huge amount of new data at every collection period and also be able to follow a reliable history of events for several years.

Another issue is the data collision. It is possible to have different tools used to make inventories, and these tools can come from different editors. This result is naming collision or erroneous data when the same thing have been probed at different times but come back for the aggregation together and have different values. Even if this case rarely happens, if it does the compliance of the Cloud is at stake.

No tools addressed Cloud environment and existing SAM tools are licensed to the number of managed devices which logically explodes if we manage clouded architectures. Besides, these tools forces us to have pre-processed data and to invest time to modify our SAM process to input the right data. Finally, as soon as we have a new *metric* or a new software, we must wait for the tool update or directly discuss with the tool editor to input our custom *metric* if we negotiated it in our contract.

In conclusion, today it is extremely difficult to have a tool that handles of those problems and manages to do perfect software management that will ensure the compliance at any time because of the above problem. The tool we created, OpTISAM, tackle this issue and handle the data in a different way of traditional tools to allow us to manage our assets in an optimized manner all the while knowing when we have a risk of noncompliance.

4.2 OpTISAM

OpTISAM is a tool designed for the Cloud nevertheless can be interfaced with standalone servers. It works by exposing several APIs allowing third-party tools to request the platform to push or pull information about SAM. These APIs allow the injection of the data necessary for the proper compliance checking and will be put in the right format by OpTISAM itself when needed. To handle the above problems about data, OpTISAM uses generic equipment definitions and aggregate data on-the-fly allowing us to store any kind of data we possess without modifying our process at all.

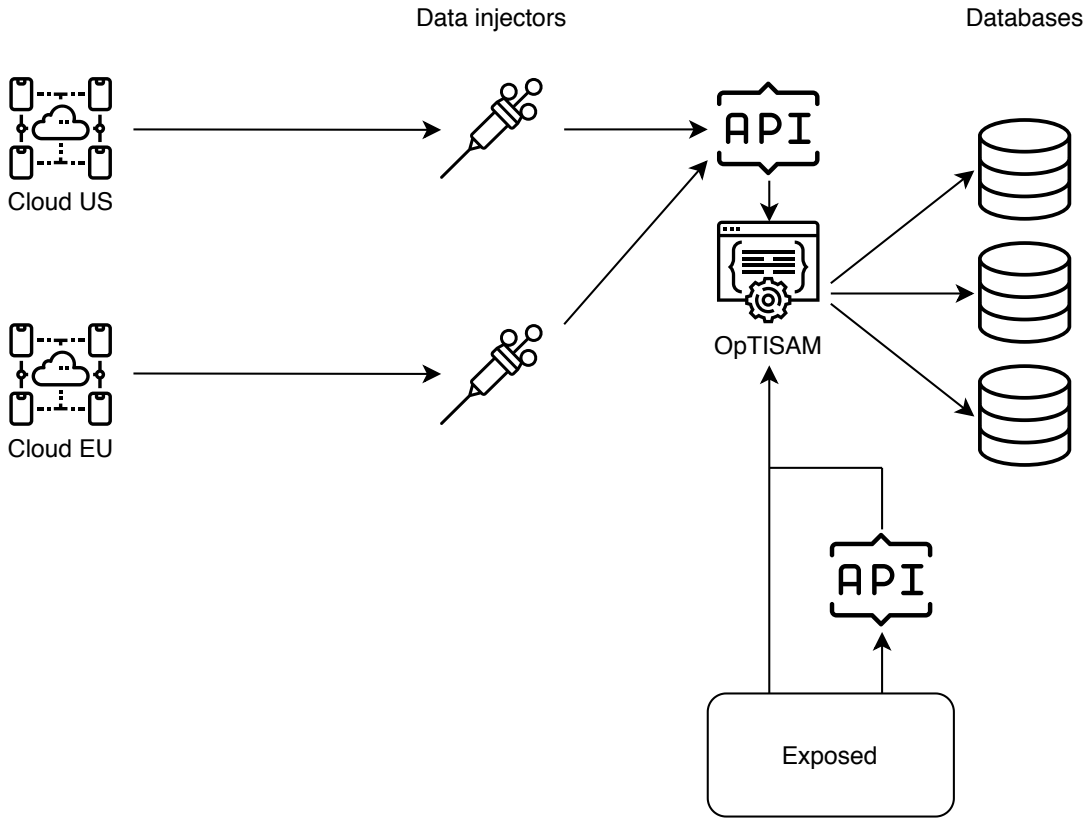


Figure 4.1: Environment of OpTISAM in the Cloud

The data is integrated via injectors as shown in the Figure 4.1. The raw data is deposited in a data lake before being used. OpTISAM uses a system of hierarchical generic equipment to handle actual and future systems. To use equipment in OpTISAM, we have to describe it via the interface. The description is composed of the name of the new equipment, its relations with other equipment and its attributes. The link with the actual data is made through the attributes. Each attribute is a link to a data source, allowing us to precisely select where do the data come from and in which format without modifying it. After the description, OpTISAM will pull from the data lake our attributes,

processing them to fit our description and will show them in the inventory page. The attributes have options to state if we want them to be shown in the equipment page, or this attribute is the identifier of the equipment. This equipment description process allow to give 'meaning' to unprocessed data coming from inventory tools. It allows also to check the health of said data by checking if some are missing or if we have duplicated data. In either case, OpTISAM will warn us that the compliance computation is not sure and will point us the source of the problem allowing us to take action and correct them before ensuring the compliance.

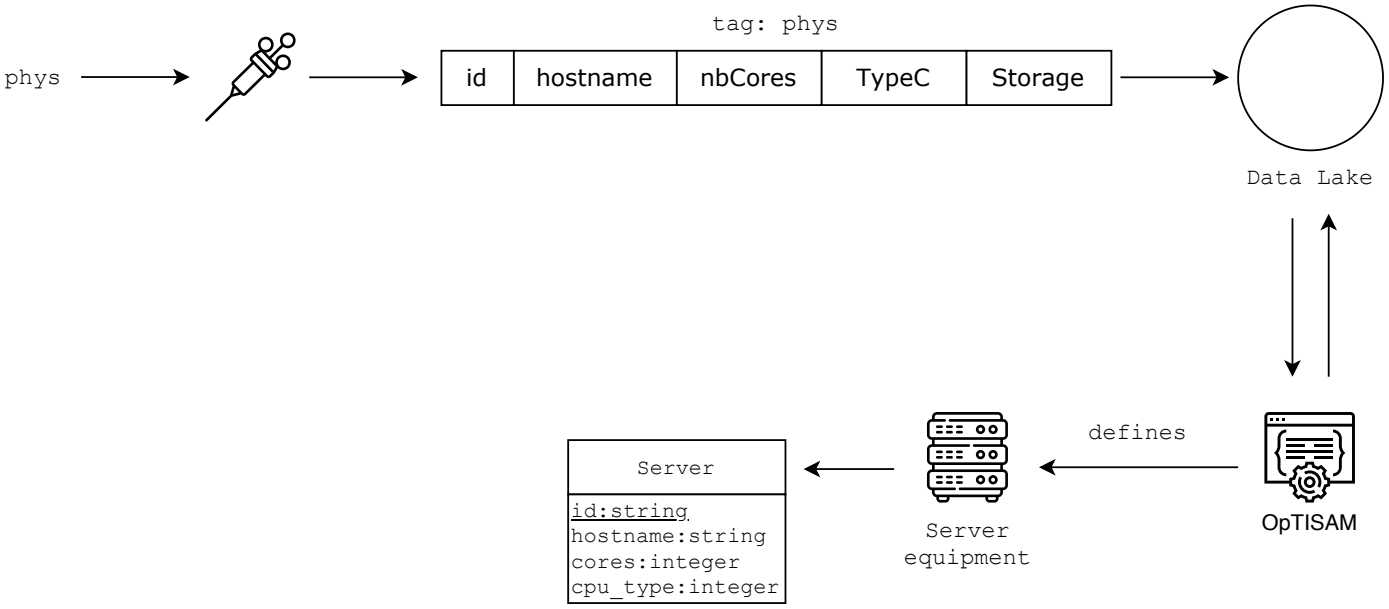
Each equipment we describe are showed in an inventory page and thanks to the relations we expressed, we can navigate from one another like searching in which Cluster a server is. If we need new attributes for a new *metric* or if the data collection process change, we can change the equipment structure in OpTISAM and the latter will properly restructure the modified equipment. For better clarity, we do an example with a data source coming from inventory tools and described in Table. 4.1. We can see that the column's names are more or less comprehensible for some (hostname, nbCores) and very difficult for others like TypeC. Understanding this data requires knowledge that the OpTISAM operator does not necessarily have. During the import of data into OpTISAM, they will be saved as such to avoid corrupting them when trying to interpret them.

Table 4.1: Data from physical servers inventory tool: 'phys'

id	*hostname*	*nbCores*	*TypeC*	*storage*
1	"server1"	8	2	16
2	"h75426-5"	16	2	16
3	"berlin"	4	3	12
4	"moscow"	32	7	12
5	"h4578-5"	4	1	16

During the creation of an equipment, here a server, we will specify which attributes we want to put in our equipment and where are they in the data source. We can see in Figure 4.2 that we ignore the last column and will give 'meaning' to columns by specifying a display name inside OpTISAM and their type. With this creation, we can now see all servers from this data source inside OpTISAM with this specific format. If later we need to know what storage is linked to the server for a *metric*, we just have to modify this equipment description and OpTISAM will adapt the data it pulls and will display the hostname and the number of cores only.

This feature allows OpTISAM to tackle the problem of *metrics* using a lot of different variables. As soon as a new *metric* use a new kind of resource (possibly every time we purchase a new contract). We can check in OpTISAM if we have this resource and if not create either a new equipment or modify one. Because of the generic equipment, the *metric* computation has to be generic too. Therefore, with the same system we have to adapt the *metric* model by linking the variable it uses to equipment attributes.



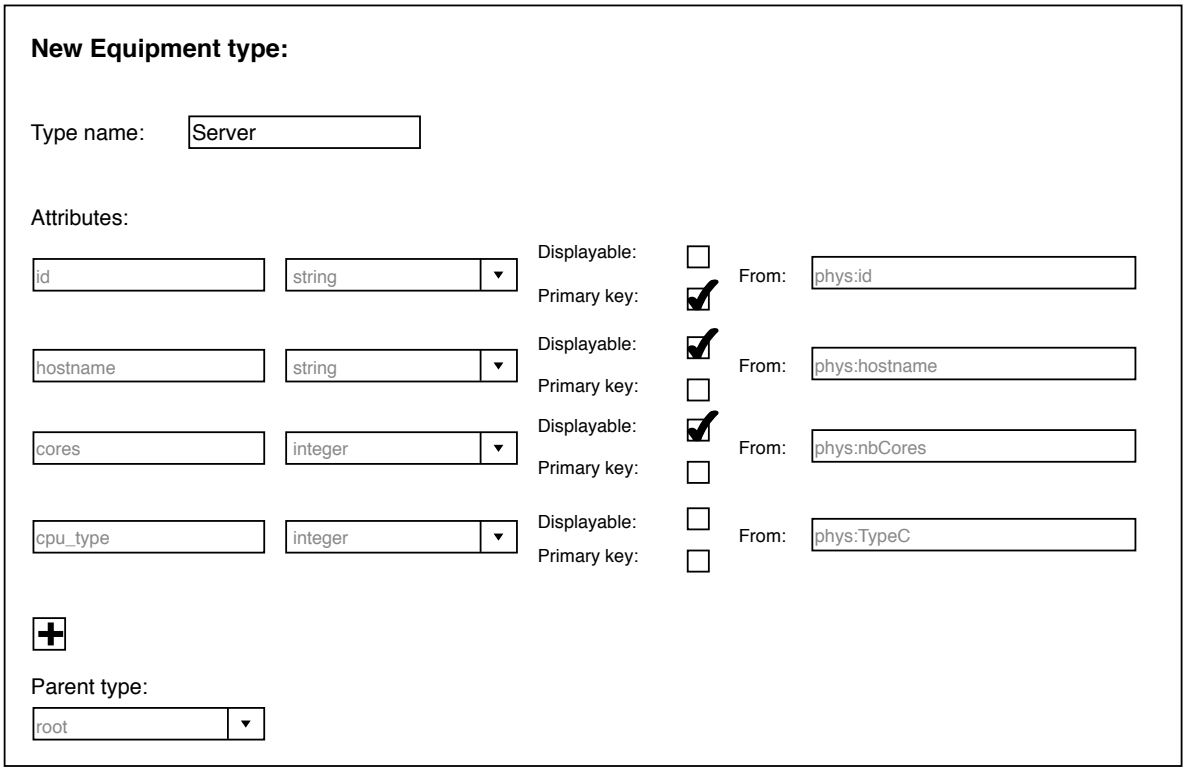
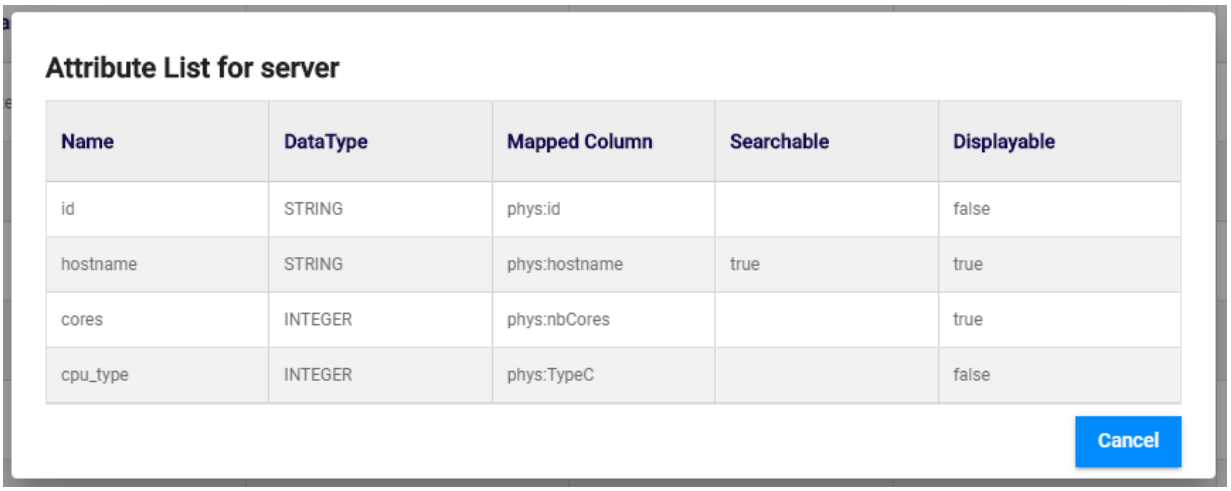


Figure 4.2: Creation example of server equipment

OpTISAM will use these attributes during the *metric* computation and will use fresh data so our compliance check is as fast as our inventory tools. It is therefore possible to modulate fast data updates for equipment that requires frequent compliance control. We can see in Figure 4.3 and Figure 4.4 the OpTISAM representation of the *phys* dataset.

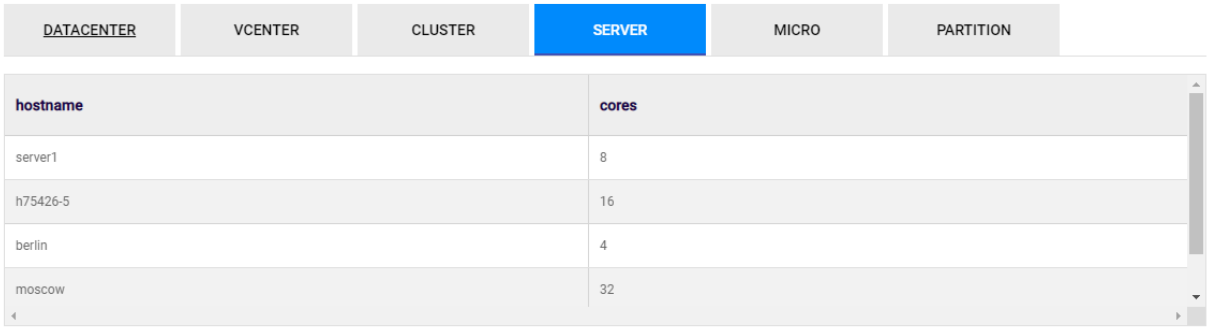


Attribute List for server

Name	DataType	Mapped Column	Searchable	Displayable
id	STRING	phys:id		false
hostname	STRING	phys:hostname	true	true
cores	INTEGER	phys:nbCores		true
cpu_type	INTEGER	phys:TypeC		false

Cancel

Figure 4.3: Attributes mapping of server equipment



DATACENTER | VCENTER | CLUSTER | SERVER | MICRO | PARTITION

hostname	cores
server1	8
h75426-5	16
berlin	4
moscow	32

Figure 4.4: Server equipment from *phys* dataset.

Combined with the proposed *metric* model presented in Chapter. 2, we are able to automate the compliance checking. If the definition of equipment changes, the *metric* can be adapted accordingly. We can see in Figure 4.5 the use of a *metric* model through OpTISAM to filter servers with equipment data and in Figure 4.6 the compliance computation validating our Adobe product licensing. This report view indicates that we are spending too much money for our licensing and that we could sell back two of our licenses.

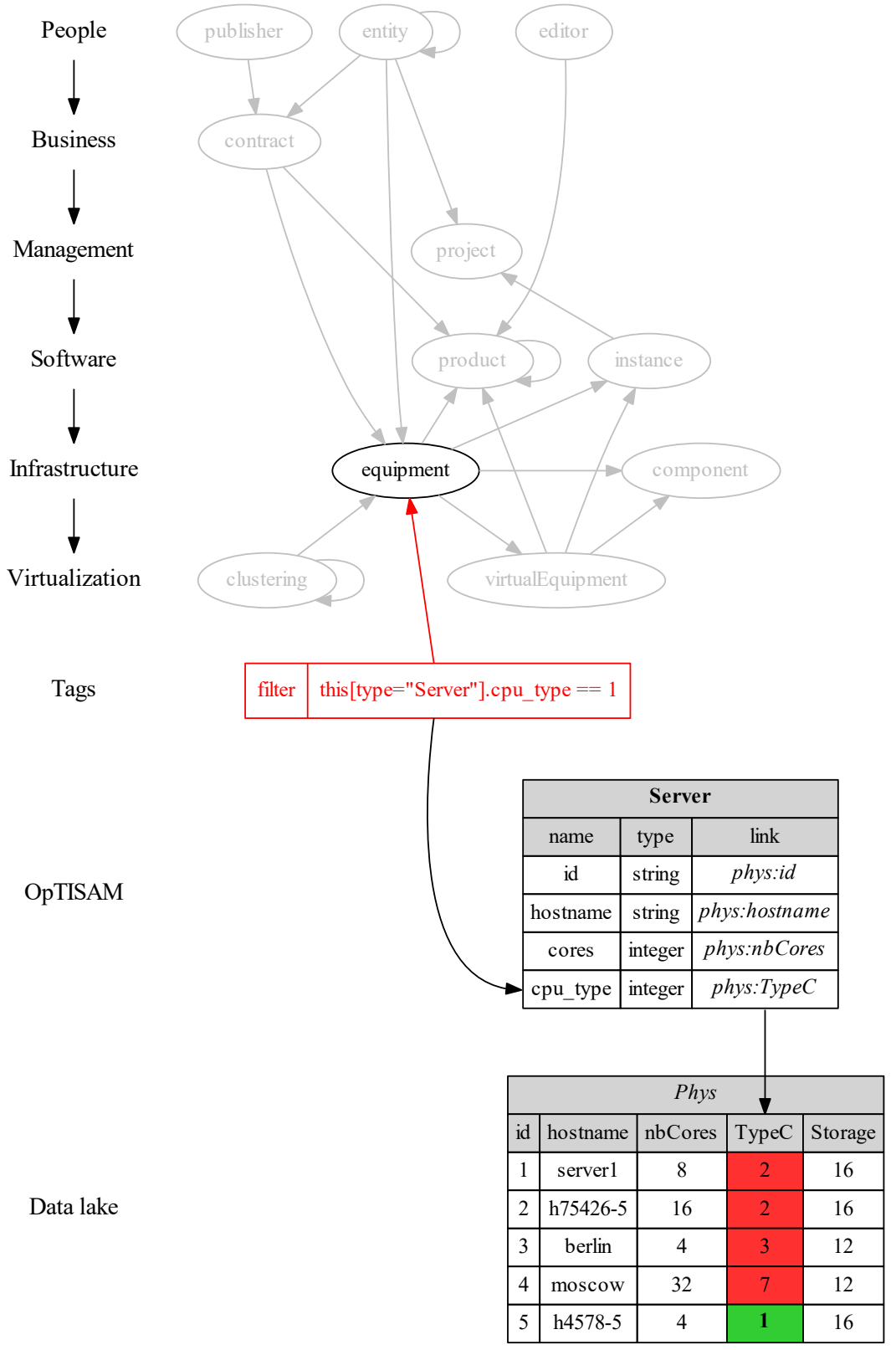


Figure 4.5: *Metric* model example use with OpTISAM

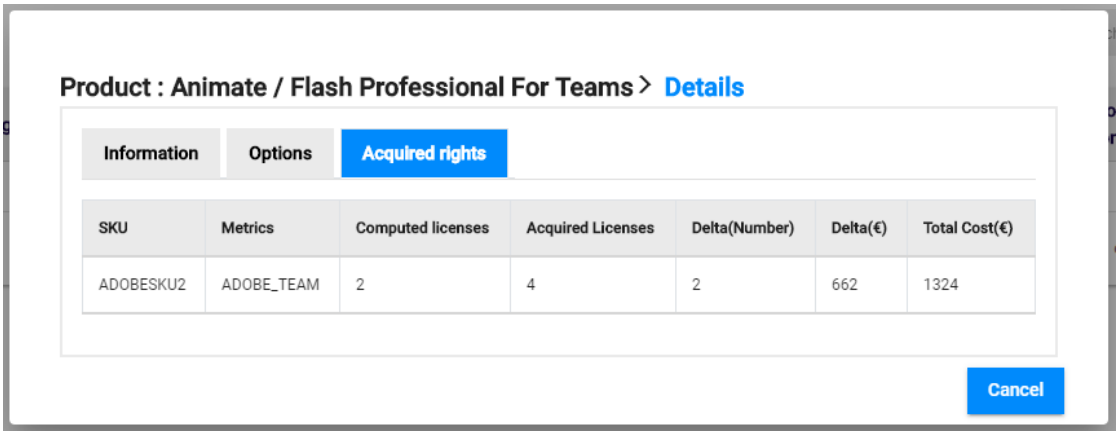


Figure 4.6: Validation of product licensing

With this kind of genericity, both for equipment and *metrics*, we enhanced the simulation features from the 2017 prototype to go further in the possibilities of forecasting. Indeed, we can now, virtually, simulate any change, anywhere and see how it would affect our compliance but even if we would be able to follow the use, or if it is a good idea to purchase a specific kind of *metric*. We also integrated the deployment algorithm we presented above and, coupled with genericity of equipment and *metric* that allows to handle any kind of present or future architecture, we can integrate OpTISAM in Cloud orchestrators such as Kubernetes [46] or ONAP [47] for virtualized network to use its deployment simulation feature to get SAM recommendations of placement.

In conclusion, OpTISAM, compared to other tools, possess highly modular handling of equipment and *metrics*. This allows to avoid touching the process in place each time we want to update our SAM with new *metric* or product. This allows to handle all kind of software as well as machine learning models or ssl certificates for example. This tool can manage up to hundreds of thousands of devices and compute their compliance in minutes compared to months with old ways. While other tools require upgrades every time a new *metric* or product appear on the market, OpTISAM requires a change of configuration in-app without any modification to the install and it will warn us if we lack data for compliance. Compared to the licensing of other tools which is based on the number of devices, OpTISAM is totally Open-Source [48] allowing the community to enhance it and integrate it virtually anywhere.

Chapter 5

Conclusion and Future works

Contents

5.1 Conclusion

In this thesis, we addressed the problem of Software Asset Management in the Cloud, and especially ensuring the compliance during the deployment of a product. Two of the main contributions of our work are a model to represent the *metric* and a deployment algorithm that allows to deploy multiple products at the same time while optimizing multiple parameters.

We introduced Software Asset Management domain and presented several problems arising with new virtualization technologies in Chapter 1. In particular, we addressed the problem of *metric* adaption in the Cloud and the lack of automation leading to problematic and time consuming compliance checks.

In Chapter 2, we proposed the *metric* model to tackle the lack of automation and confronted it to several use cases to show its usability and efficiency compared to old methods.

The main focus of this thesis was on the deployment optimization and compliance ensurance presented in Chapter 3. We started by showing the impact a good deployment algorithm could have on SAM considerations. We then introduced a new kind of heuristic oriented towards SAM aspects that optimizes several parameters during the deployment of a product on a Cloud.

This algorithm showed promising results and as we wanted to push it further, we proposed another one that finds a near optimal deployment of multiple products in one computation. We proved that this decision problem was NP-Complete and showed that this algorithm obtains really good results with both low memory and low time allocated.

Finally, in Chapter 4, we enhanced an existing tool, OpTISAM, with these new

algorithms and models to propose a highly modular and generic tool that can manage any type of equipment with any kind of *metric* and optimize the placement of products on the Cloud it manages.

From an academic point of view, we have advanced the state-of-the-art by bringing a full automation of the SAM processes and with it new algorithms for multi-parametric product deployment on Clouded architectures. We proved the NP-completeness of SAM aware deployment and proposed a near-optimal heuristic that solve this problem.

From an industrial point of view, we solved a rising problem among companies with SAM needs. The proof of NP-Completeness showed that this problem was not easy and that without automation, SAM teams could not ensure efficiently their software park. This automation allows performant and reliable compliance check, removes human error from the equation and is modular enough to be adapted to future technologies. Besides, the upgrade of OpTISAM with genericity and a new deployment algorithm allows them to tackle SAM constraints without problems.

5.2 Future works

Future work concerns a more in-depth analysis of certain discoveries, new proposals to try different methods or simply curiosity about the behaviour of the SAM in other areas. There is a lot of ideas that we would have liked to investigate in the end of this Thesis:

First, it would be interesting to see how well the *metric* model integrate in ISO 19770 [10] norm and ODRL [35]. With this integration, we could be able to change *metric* on the fly to another that suits our needs following a deployment for example. We could even try to integrate this model inside smart contracts to enhance the speed of automatic detection and optimization of software cost. Coupling the model with ISO or ODRL will allow us to automate the discussion with editors about SAM considerations and could bring pre-test of software to see if the proposed *metrics* are 'dangerous' or not with our current usage. Then, it would be interesting to see how this model behave with new kind of *metrics* coming with the rise of SDN/NFV (network virtualization). For example, SAU: We have to compute the number of devices connected per antennas and do the average for every hour. Then, for each hour, we have to compute the sum of all average for that hour. Finally, the number of licenses to possess is the maximum of the sum computed over 24h. This *metric* is complicated to compute and, therefore, its adaptation to the *metric* model will present some interesting challenges.

In a second step, it would be interesting to add more criteria to the heuristic and, therefore, enhance the deployment algorithm. We could think of adding network aware criterion or more industrial criterion like the support level needed for the product to deploy.

Following these considerations, it would be interesting to find a way to do real energy and performance variables computation during the deployment to have a very precise state of each server. Currently, these variables are inputed by algorithm users but could be computed with state of the art third-party algorithm.

Thirdly, it would be very interesting to enhance the heuristic either by adding *metric*

aware formula or simply enhancing the current one. Currently, following some experiments, the performance criteria is more taken into account than the other. Maybe, modifying the formula could bring even better results that the current ones. The first option, adding a *metric* aware formula, is adapting the node traversal computation depending on the current product *metric*. For example, the Oracle Database Processor *metric* will cost 0 licenses if there is already databases on the cluster. This information used by the first algorithm we presented for deployment lead to significant optimization at single product level. Maybe by exploring more *metrics* optimization, we could enhance the multiple product deployment by using local optimization when we can.

Concerning OpTISAM, some work could be done concerning the network saturation induced by the amount of data converging towards the data lake. We could search some SAM aware filters to avoid storing all data or solve some data problems we mentioned earlier. Next, the integration of OpTISAM to some orchestrators like ONAP for network or Kubernetes could be very informative about the current compliance state in near real-time using the orchetrastor information. In the other way, OpTISAM could bring SAM considerations to deployment algorithm of such orchestrators.

Another breakthrough could be the addition of usable forecasting in the SAM field. We introduced the problem at the end of Chapter 3 and by using state of the art data science and machine learning, we could help forecasting future noncompliance because of unplanned events like unexpected shutdown with intelligent predictive maintenance. AI is admittedly difficult to integrate to all SAM processes but could be used in licenses stock threshold detection.

Finally, a more theoretical thought about SAM in general is that it should integrate software engineering considerations. Indeed, knowing how a product have been developed or how it behave could bring many optimization about its deployment or parameters optimization such as energy (some works have been done on energy consumption relation to development). Besides, adding SAM in the process of continuous integration could allow SAM operators to quickly have the information about software bugs or security breaches. This could also allow SAM operators to give hints about product selection for a project. For example, if a software development team should choose between different proprietary databases, maybe one would be more expensive for development but, in production, would be cheaper than the other one because of *metric* optimization.

Appendix A

License Metric Language grammar

Regex defined tokens:

```
CONSTANT = 0[xX][a-fA-F0-9]+ | [0-9]+ | [0-9]+[Ee][+-]?[0-9]+
          | [0-9]*\.[0-9]+([Ee][+-]?[0-9])? | [0-9]+\.[0-9]*([Ee][+-]?[0-9])?
STRING_LITERAL = [a-zA-Z_]?\'(\\.|[^\\\'])+\' | [a-zA-Z_]?"(\\.|[^\\"])*"
RIGHT_OP = >>
LEFT_OP = <<
AND_OP = &
OR_OP = \|
LE_OP = <=
GE_OP = >=
EQ_OP = ==
NE_OP = !=
COMMA = ,
ASSIGN = =
L_PAREN = \(
R_PAREN = \)
L_BRACKET = \[
R_BRACKET = \]
DOT = \.
AND = &&
NEG = !
SUB = -
ADD = \+
MUL = \*
DIV = //
SUBNODE = /
MOD = %
INF = <
```

```
SUP = >
POW = \^
OR = \|\|
EXT = ext
THIS = this
ID = [a-zA-Z_]([a-zA-Z_]|[0-9])*
```

Grammar:

⟨*statement*⟩ ::= ⟨*mutation*⟩
| ⟨*comparison*⟩
| ⟨*function*⟩
| ⟨*attr*⟩
| ⟨*operation*⟩

⟨*mutation*⟩ ::= ⟨*attr*⟩ ASSIGN ⟨*operation*⟩
| ⟨*attr*⟩ ASSIGN ⟨*function*⟩

⟨*operation*⟩ ::= ⟨*operation_arg*⟩ ⟨*op*⟩ ⟨*operation_arg*⟩
| ⟨*operation*⟩ ⟨*op*⟩ ⟨*operation_arg*⟩

⟨*op*⟩ ::= ADD
| MUL
| DIV
| SUB
| POW
| POW
| LEFT_OP
| RIGHT_OP
| OR_OP
| AND_OP
| MOD

⟨*operation_arg*⟩ ::= ⟨*attr*⟩
| ⟨*val*⟩
| ⟨*function*⟩

⟨*function*⟩ ::= EXT DOT ⟨*function_external_long*⟩
| ID L_PAREN ⟨*func_args*⟩ R_PAREN
| ID L_PAREN R_PAREN

⟨*function_external_long*⟩ ::= ⟨*function*⟩
| ID DOT ⟨*function_external_long*⟩

⟨*func_args*⟩ ::= ⟨*possible_arg*⟩
| ⟨*possible_arg*⟩ COMMA ⟨*func_args*⟩

⟨*possible_arg*⟩ ::= ⟨*attr*⟩
| ⟨*val*⟩
| ⟨*path*⟩
| ⟨*function*⟩

⟨*comparison*⟩ ::= ⟨*comparison_set*⟩
| ⟨*comparison_set*⟩ AND ⟨*comparison*⟩
| ⟨*comparison_set*⟩ OR ⟨*comparison*⟩

⟨*comparison_set*⟩ ::= ⟨*comparison_arg*⟩ ⟨*comp_op*⟩ ⟨*comparison_arg*⟩

⟨*comparison_arg*⟩ ::= ⟨*attr*⟩
| ⟨*val*⟩
| ⟨*function*⟩
| ⟨*operation*⟩

⟨*attr*⟩ ::= ⟨*path*⟩ DOT ID
| NEG ⟨*path*⟩ DOT ID

⟨*path*⟩ ::= ⟨*node*⟩
| ⟨*path*⟩ SUBNODE ⟨*node*⟩

⟨*node*⟩ ::= ID
| THIS
| THIS ⟨*filter*⟩
| ID ⟨*filter*⟩

⟨*filter*⟩ ::= L_BRACKET ⟨*filter_comparison_args*⟩ R_BRACKET

⟨*filter_comparison_args*⟩ ::= ⟨*filter_comparison*⟩
| ⟨*filter_comparison*⟩ COMMA ⟨*filter_comparison_args*⟩

⟨*filter_comparison*⟩ ::= ⟨*filter_comparison_set*⟩
| ⟨*filter_comparison_set*⟩ OR ⟨*filter_comparison*⟩

⟨*filter_comparison_set*⟩ ::= ID ⟨*comp_op_filter*⟩ ⟨*val*⟩

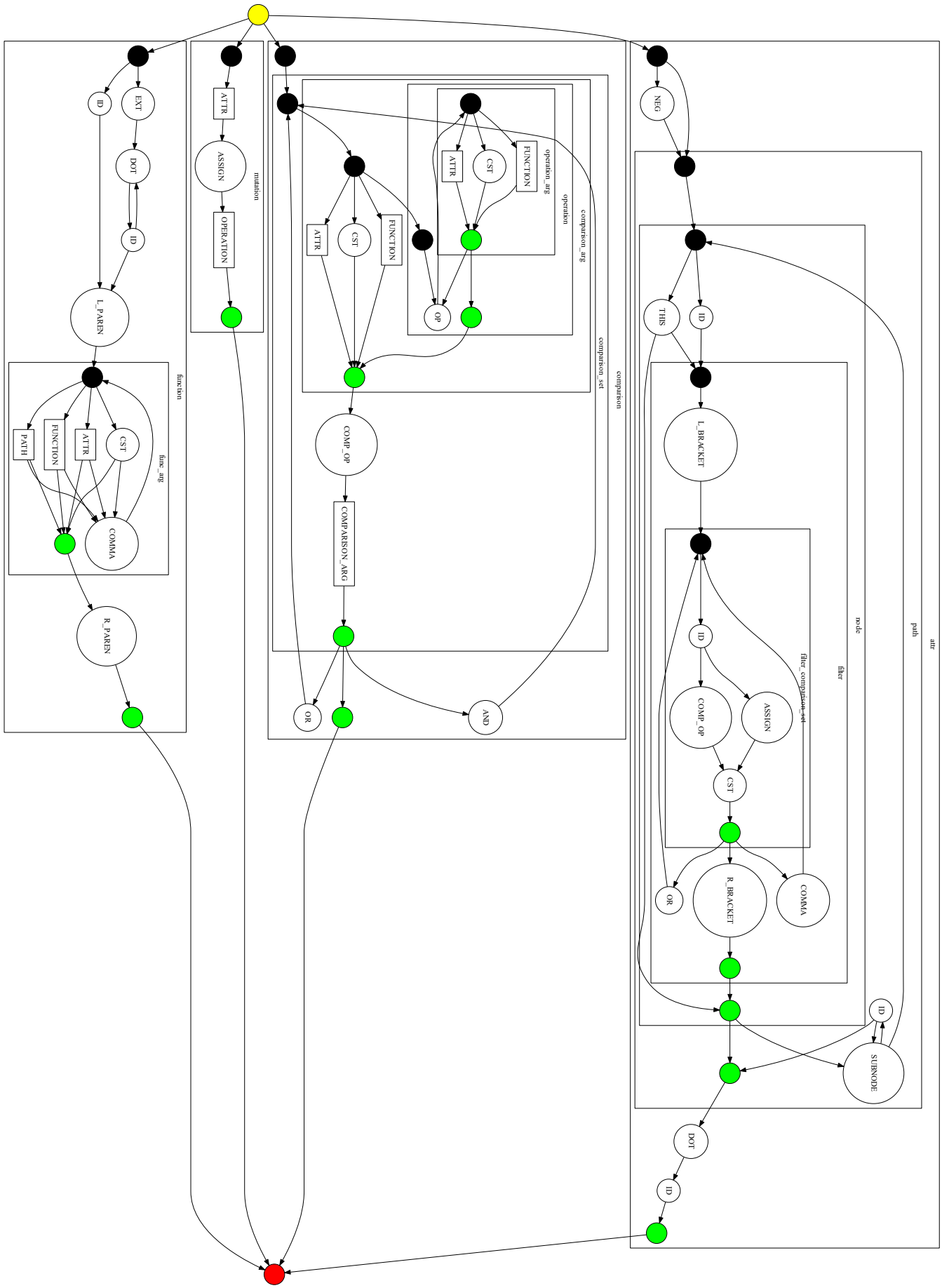
⟨*comp_op_filter*⟩ ::= ⟨*comp_op*⟩
| ASSIGN

⟨*comp_op*⟩ ::= LE_OP
| GE_OP
| EQ_OP
| NE_OP
| INF
| SUP

⟨*val*⟩ ::= STRING_LITERAL
| CONSTANT

Appendix B

Finite-State Machine of LML


attr
path
NEG
node
THIS
ID
L_BRACKET
filter
filter_comparison_set
ID
COMP_OP
ASSIGN
CST
OR
R_BRACKET
COMMA
ID
SUBNODE
DOT
ID
comparison
comparison_set
operation
operation_arg
ATTR
CST
FUNCTION
OP
comparison_arg
ATTR
CST
FUNCTION
COMP_OP
COMPARISON_ARG
OR
AND
mutation
ATTR
ASSIGN
OPERATION
function
ID
EXT
DOT
ID
L_PAREN
func_arg
PATH
FUNCTION
ATTR
CST
COMMA
R_PAREN

List of Figures

Bibliography

[1] *SAP UK Ltd v Diageo Great Britain Ltd [2017] EWHC 189 (TCC)*. Feb. 2017. URL: http://www.bailii.org/ew/cases/EWHC/TCC/2017/189.html.

[2] Peter Sayer. *SAP settles licensing dispute with AB InBev*. Mar. 2018. URL: https://www.itworld.com/article/3264435/sap-settles-licensing-dispute-with-ab-inbev.html.

[3] Microsoft. *Microsoft license terms for Windows operating system (2.c.v)*. June 2018. URL: https://www.microsoft.com/en-us/Useterms/Retail/Windows/10/UseTerms_Retail_Windows_10_English.htm.

[4] Oracle. *Database Licensing*. June 2020. URL: https://www.oracle.com/assets/databaselicensing-070584.pdf.

[5] Oracle. *Oracle Processor Core Factor Table*. Oct. 2019. URL: http://www.oracle.com/us/corporate/contracts/processor-core-factor-table-070634.pdf.

[6] Oracle. *Oracle Technology Global Price List*. Jan. 2020. URL: https://www.oracle.com/assets/technology-price-list-070617.pdf.

[7] SAP SE or an SAP affiliate company. *Indirect Access Guide for SAP Installed Base Customers*. Apr. 2018. URL: https://news.sap.com/wp-content/blogs.dir/1/files/Indirect_Access_Guide_for_SAP_Installed_Base.pdf.

[8] Amazon Web Services. *Oracle Licensing Considerations*. May 2020. URL: https://docs.aws.amazon.com/whitepapers/latest/oracle-database-aws-best-practices/oracle-licensing-considerations.html.

[9] *Migration Complete – Amazon's Consumer Business Just Turned off its Final Oracle Database*. Oct. 2019. URL: https://aws.amazon.com/fr/blogs/aws/migration-complete-amazons-consumer-business-just-turned-off-its-final-oracle-database/.

[10] International Organization for Standardization. *ISO/IEC 19770-1:2017(en)*. 2017. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:19770:-1:ed-3:v1:en.

[11] M. Ben-Menachem and G.S. Marliss. "IT Assets—Control by Importance and Exception: Supporting the "Paradigm of Change"." In: *IEEE Software* 22.4 (July 2005), pp. 94–102. DOI: 10.1109/ms.2005.99. URL: https://doi.org/10.1109/ms.2005.99.

[12] M.F. Bott. "Software as a corporate asset." In: *IEE Proceedings - Software* 147.2 (2000), p. 31. DOI: 10.1049/ip-sen:20000600. URL: https://doi.org/10.1049/ip-sen:20000600.

[13] M. Ben-Menachem and G.S. Marliss. "Inventorying Information Technology Systems: Supporting the "Paradigm of Change"." In: *IEEE Software* 21.05 (Sept. 2004), pp. 34–43. DOI: 10.1109/ms.2004.1331300. URL: https://doi.org/10.1109/ms.2004.1331300.

[14] Benno E. Albert, Rodrigo P. dos Santos, and Claudia M. Werner. "Software ecosystems governance to enable IT architecture based on software asset management." In: *2013 7th IEEE International Conference on Digital Ecosystems and Technologies (DEST)*. IEEE, July 2013. DOI: 10.1109/dest.2013.6611329. URL: https://doi.org/10.1109/dest.2013.6611329.

[15] Jody Swartz and Paulius Vysniauskas. "Software Asset Management in Large Scale Organizations- Exploring the Challenges and Benefits." MA thesis. University of Gothenburg, Mar. 2015.

[16] Pete Rotella. "Software security vulnerabilities." In: *Proceedings of the 1st International Workshop on Security Awareness from Design to Deployment - SEAD '18*. ACM Press, 2018. DOI: 10.1145/3194707.3194708. URL: https://doi.org/10.1145/3194707.3194708.

[17] Jeffrey Voas and George Hurlburt. "Third-Party Software's Trust Quagmire." In: *Computer* 48.12 (Dec. 2015), pp. 80–87. DOI: 10.1109/mc.2015.372. URL: https://doi.org/10.1109/mc.2015.372.

[18] Chris Williams. *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript*. Mar. 2016. URL: https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/.

[19] Rajiv D. Banker and Robert J. Kauffman. "Automated Software Metrics, Repository Evaluation and Software Asset Management: New Tools and Perspectives for Managing Integrated Computer Aided Software Engineering (I-Case)." In: *Information Systems Working Papers Series, Vol* (1991).

[20] Neil F. Holsing and Davidc. Yen. "Software Asset Management." In: *Information Resources Management Journal* 12.3 (July 1999), pp. 14–26. DOI: 10.4018/irmj.1999070102. URL: https://doi.org/10.4018/irmj.1999070102.

[21] P.K. Gocek et al. *Obtaining software asset insight by analyzing collected metrics using analytic services*. US Patent 9,652,812. May 2017. URL: https://www.google.com/patents/US9652812.

[22] Noëlle Baillon et al. "Software license optimization and cloud computing." In: *CLOUD COMPUTING 2017* (2017), p. 125.

[23] Kelley Dempsey et al. *Automation Support for Security Control Assessments: Software Asset Management*. Tech. rep. National Institute of Standards and Technology, 2018.

[24] Joseph Elias Mbowe et al. “A Conceptual Framework for Threat Assessment Based on Organization’s Information Security Policy.” In: *Journal of Information Security* 05.04 (2014), pp. 166–177. DOI: 10.4236/jis.2014.54016. URL: https://doi.org/10.4236/jis.2014.54016.

[25] Flexera. *How Security Risks & the Shift to the Cloud Are Transforming SAM*. Tech. rep. 2016. URL: https://resources.flexera.com/web/pdf/WhitePaper-SLO-Security-Risks-Cloud-Transforming-SAM.pdf.

[26] Ana Márcia Quitério Varela, Mirian Picinini Méxas, and Geisa Meirelles Drumond. “The scenario of software asset management (SAM) in large and midsize companies.” In: *Independent Journal of Management & Production* 9.2 (June 2018), p. 301. DOI: 10.14807/ijmp.v9i2.730. URL: https://doi.org/10.14807/ijmp.v9i2.730.

[27] Paul Klint and Chris Verhoef. “Enabling the creation of knowledge about software assets.” In: *Data & Knowledge Engineering* 41.2-3 (June 2002), pp. 141–158. DOI: 10.1016/s0169-023x(02)00038-1. URL: https://doi.org/10.1016/s0169-023x(02)00038-1.

[28] Mordechai Ben-Menachem. “Towards management of software as assets: A literature review with additional sources.” In: *Information and Software Technology* 50.4 (Mar. 2008), pp. 241–258. DOI: 10.1016/j.infsof.2007.08.001. URL: https://doi.org/10.1016/j.infsof.2007.08.001.

[29] Martin Jakubička. “Software asset management.” In: *2010 IEEE International Conference on Software Maintenance*. IEEE, Sept. 2010. DOI: 10.1109/icsm.2010.5609662. URL: https://doi.org/10.1109/icsm.2010.5609662.

[30] Fedor Dzerzhinskiy. “About Lawyers, Programmers, and Software Assets.” In: Mar. 2012. DOI: 10.13140/RG.2.1.3039.2488. URL: https://doi.org/10.13140/RG.2.1.3039.2488.

[31] Gary Stoneburner, Alice Y. Goguen, and Alexis Feringa. *Risk Management Guide for Information Technology Systems*. Tech. rep. National Institute of Standards and Technology, 2002.

[32] Microsoft. *Introduction to Per Core Licensing and Basic Definitions*. May 2020. URL: https://download.microsoft.com/download/3/d/4/3d42bdc2-6725-4b29-b75a-a5b04179958b/percorelicensing_definitions_vlbrief.pdf.

[33] *Dgraph database*. Apr. 2019. URL: https://dgraph.io/.

[34] *Technologies de l’information — Gestion de biens de logiciel — Partie 3: Schéma de droit de logiciel*. URL: https://www.iso.org/fr/standard/52293.html.

[35] *ODRL Information Model 2.2*. Feb. 2018. URL: https://www.w3.org/TR/odrl-model/.

[36] Ian Foster and Carl Kesselman. "Computational Grids." In: *Vector and Parallel Processing — VECPAR 2000*. Ed. by José M. L. M. Palma, Jack Dongarra, and Vicente Hernández. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 3–37. ISBN: 978-3-540-44942-3.

[37] Jack Dongarra et al. "The International Exascale Software Project roadmap." In: *The International Journal of High Performance Computing Applications* 25.1 (Jan. 2011), pp. 3–60. DOI: 10.1177/1094342010391989. URL: https://doi.org/10.1177/1094342010391989.

[38] A. Berl et al. "Energy-Efficient Cloud Computing." In: *The Computer Journal* 53.7 (Aug. 2009), pp. 1045–1051. DOI: 10.1093/comjnl/bxp080. URL: https://doi.org/10.1093/comjnl/bxp080.

[39] Daniel Balouek-Thomert, Eddy Caron, and Laurent Lefevre. "Energy-Aware Server Provisioning by Introducing Middleware-Level Dynamic Green Scheduling." In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, May 2015. DOI: 10.1109/ipdpsw.2015.121. URL: https://doi.org/10.1109/ipdpsw.2015.121.

[40] Orgerie Anne-Cécile and Lefèvre Laurent. "When Clouds become Green: the Green Open Cloud Architecture." In: *Advances in Parallel Computing* 19.Parallel Computing: From Multicores and GPU's to Petascale (2010), pp. 228–237. ISSN: 0927-5452. DOI: 10.3233/978-1-60750-530-3-228. URL: http://doi.org/10.3233/978-1-60750-530-3-228.

[41] Zoltan Adam Mann. "Resource Optimization Across the Cloud Stack." In: *IEEE Transactions on Parallel and Distributed Systems* 29.1 (Jan. 2018), pp. 169–182. DOI: 10.1109/tpds.2017.2744627. URL: https://doi.org/10.1109/tpds.2017.2744627.

[42] *SmartTrack : Gestion des licences logicielles*. May 2020. URL: https://www.aspera.com/fr/gestion-des-licences/smarttrack/.

[43] *Snow License Manager*. May 2020. URL: https://www.snowsoftware.com/int/products/snow-license-manager.

[44] *FlexNet Manager*. May 2020. URL: https://www.flexnetmanager.com/.

[45] Anne-Lucie Vion. "Software Asset Management and Cloud Computing. (Gestion du patrimoine logiciel et Cloud Computing)." PhD thesis. Grenoble Alpes University, France, 2018. URL: https://tel.archives-ouvertes.fr/tel-01901991.

[46] *Kubernetes*. May 2020. URL: https://kubernetes.io/.

[47] *Open Network Automation Platform*. May 2020. URL: https://www.onap.org/.

[48] *OpTISAM GitHub*. May 2020. URL: https://github.com/Orange-OpenSource/optisam-backend.