TÉCNICO LISBOA

Lynceus: Long-Sighted, Budget-Aware Online Tuning of Cloud Applications

Maria da Loura Casimiro

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. João Nuno de Oliveira e Silva
Prof. Paolo Romano

Examination Committee

Chairperson: Prof. António Manuel Raminhos Cordeiro Grilo
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

November 2018

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgements

Firstly, I thank my advisors, Professor Paolo Romano and Professor João Nuno Silva, for welcoming me as their student and for their guidance and support throughout this work. Specially, I kindly thank Professor Paolo for setting up weekly meetings to ensure a continuous and steady progress.

Secondly, I thank Professor Luís Rodrigues, for introducing me to Professor Paolo and for making this collaboration possible, and Diego Didona, who worked relentlessly towards Lynceus' success and who was always available for skype calls and to clarify doubts.

I thank my family for the endless support, encouragement and opportunities throughout my studies and my life.

Lastly, I thank all my friends and my colleagues from GSD, particularly, from room 501, who became good friends and without whom this process would have been much harder. A special thank you to Diogo Barradas, who read everything that I wrote countless times, who was rooting for my success from the very beginning, and for all the precious help gathering Lynceus' datasets, along side Manuel Reis and Pedro Joaquim.

Lisbon, November 2018
Maria da Loura Casimiro

For my parents,

Resumo

Os fornecedores de serviços de computação na nuvem têm vindo a aumentar a diversidade das plataformas que disponibilizam, nomeadamente em termos de poder de cálculo, quantidade de memória, capacidade de armazenamento e de largura de banda da rede. Por um lado, esta diversidade oferece mais opções ao utilizador mas, por outro, torna a tarefa de escolher a configuração certa muito mais complexa, visto que configurações com desempenho semelhante podem ter custos muito diferentes, e nem sempre é fácil antecipar qual a configuração mais barata que satisfaz os requisitos da aplicação. Neste contexto, o estudo de técnicas que permitam automatizar o processo de seleção da melhor configuração para executar uma dada aplicação na nuvem tem vindo a ganhar relevo.

As abordagens recentes para identificar a configuração ótima para correr trabalhos na nuvem baseiam-se numa fase de exploração durante a qual a aplicação é executada num conjunto diverso de configurações. Estes sistemas conseguem encontrar a configuração próxima da ótima. No entanto, não consideram o custo da fase de exploração, que pode ser significativo.

Esta dissertação estuda técnicas de procura que têm em conta o custo de exploração. Propomos um algoritmo que permite reduzir o custo da exploração através de uma criteriosa escolha das configurações a experimentar, que tem em conta o custo de cada experiência e a contribuição esperada da mesma para a exatidão do modelo que prevê as próximas explorações. Os resultados da nossa avaliação mostram que a solução proposta é capaz de encontrar configurações perto da ótima com baixo custo.

Abstract

In recent years we have witnessed a trend for the cloud providers to increase the diversity of the platforms they offer, namely with respect to computational power, memory, storage capacity and network bandwidth. If it's true that this diversity allows users to choose amongst a broader set, it is also a true that such diversity renders the process of selecting the right configuration much more complex. This happens because configurations with similar performance may possess very different costs. Moreover, it is not always easy to anticipate which is the cheaper configuration that complies with the application requirements. Thus, the study of techniques for automating the process of selecting the best configuration to execute cloud applications has gained relevance.

Recent approaches to identify the optimal configuration to execute cloud applications are based on an exploration phase, during which the application is executed in a set of configurations. These systems have been shown to identify near-optimal configurations. However, the cost of the exploration phase, which can be rather high, is not taken into account.

This dissertation studies search techniques that consider the cost of the exploration phase. We propose an algorithm which allows to reduce the cost of the exploration phase through a judicious choice of the configurations to explore. This choice accounts both for the cost of each experiment and the expected improvement they will bring to the predictive model that guides the exploration. Our evaluation results show that our algorithm is able to identify near-optimal configurations at a low cost.

Palavras Chave
Keywords

Palavras Chave

Computação na Nuvem; Custo de Exploração; Custo de Execução; Configuração ótima; Máquina Virtual

Keywords

Cloud Computing; Exploration Cost; Exploitation Cost; Optimal Configuration; Virtual Machine

Contents

List of Figures

List of Tables

Acronyms

AWS Amazon Web Services

BO Bayesian Optimization

CDF Comulative Distribution Function

CF Collaborative Filtering

CNN Convolutional Neural Network

DFO Distance From Optimum

DP Dynamic Programming

EC2 Elastic Compute Cloud

EI Expected Improvement

EIc constrained Expected Improvement

GCE Google Compute Engine

GP Gaussian Process

KPI Key Performance Indicator

LCB Lower Confidence Bound

LHS Latin Hyper-Cube Sampling

LSTM Long Short-Term Memory

ML Machine Learning

NEX Number of Explorations

NN Neural Network

OED Optimal Experimental Design

PDF Probability Distribution Function

PI Probability of Improvement

QoS Quality of Service

RNN Recurrent Neural Network

RS Recommender System

SMBO Sequential Model Based Optimization

TM Transactional Memory

UCB Upper Confidence Bound

vCPU virtual CPU

VM Virtual Machine

1 Introduction

Cloud computing is an abstraction that allows users to deploy, configure, and execute services on shared pools of resources, which can be rented according to the needs of each application. Many of the services provided over the internet, such as sending e-mails, editing documents, listening to music or watching TV are now supported by cloud computing. Cloud computing services started to be offered by Amazon in 2006 and, since then, have been increasingly adopted by public and private organization worldwide.

Amazon (with Amazon Web Services (AWS) Elastic Compute Cloud (EC2) [3]), Google (with Google Compute Engine (GCE) [31]), and Microsoft with Microsoft Azure [48], are some of the main current cloud computing providers. They offer a wide range of platforms and virtual machine types, that vary in terms of computational, memory, and network capacities. When deploying an application in the cloud, it is critical to select the appropriate resources. For instance, running a memory intensive job in a machine that does not have enough memory will lead to poor performance, regardless of the CPU capacity. Unfortunately, it is not always easy to anticipate which is the cheaper configuration that complies with the application requirements. Thus, the study of techniques for automating the process of selecting the best configuration to execute cloud applications has gained significant relevance in this context.

1.1 Motivation

A large number of research efforts [66, 2, 69, 35, 21, 20, 66, 15, 18] have addressed the problem of selecting the best configuration for deploying a given application in the cloud. A configuration consists of a set of virtual machines' parameters, such as the number of Virtual Machines (VMs) and their sizes, and of a set of application specific parameters which, for example in the case of a neural network training job, may be the learning rate or the batch size.

Existing approaches target different objective functions (e.g., minimizing user [66, 2, 69, 35] vs provider [21, 20, 66] costs) and employ a wide range of predictive techniques. Despite their differences, though, they share a key common mechanism: an exploratory phase during which the target application is deployed and tested over a diverse set of configurations in order to build a model that maps the possible system's configurations to the corresponding application's performance. Which and how many configurations will have to be explored before a final recommendation for the system's configuration is outputted is typically established in a dynamic fashion, based on the shape of the performance function over which the model is being fitted and on the expected accuracy of the model learnt so far.

State-of-the-art systems have been shown to be able to identify near-optimal configurations for the final application deployment (i.e., its steady state). Unfortunately, existing solutions aim solely at optimizing the efficiency of the final configuration. As such, they neglect the cost of the exploration phase. As a matter of fact, the cost of the exploration phase can be quite expensive not only for short running jobs, but also for long running applications that are subject to frequent workload changes. In this

case, the system has to undergo frequent re-optimization phases in order to adapt to workload changes and pursue optimal efficiency.

Since existing systems are mainly focused on optimizing performance and minimizing deployment cost, usually at the expense of the exploration cost, a problem remains: what configuration provides users with a workload performance above some desirable threshold while maintaining both the deployment cost and the cost for finding that solution below some desirable limit?

1.2 Objectives and Proposed Methodology

The goal of this work is to build a self-tuning system for cloud applications that aims to optimize the cost efficiency not only of the final system's configuration but also of the exploration phase performed as part of the tuning process, while ensuring the user imposed Quality of Service (QoS) restrictions (such as the maximum running time for a job) are complied with. This is a non-trivial problem as the exploration cost depends not only on the configurations that are being tested but also on the duration of the test. Furthermore, the configurations that are explored influence how the model is updated and therefore modify the next choices/predictions of configurations to explore. Exploring configurations that are both cheap and provide good workload performance is paramount for the improvement of the model, for the discovery of the optimal configuration and for the reduction of the overall costs.

We argue that, in order to tackle this problem, the optimization process should explicitly take into account the cost dynamics of the exploration phase. Current state-of-the-art systems reason/plan the next exploration steps using a greedy strategy that only looks at the immediate expected "reward" from visiting a given configuration, say c. Instead of considering only the expected reward, in our work we also consider the expected cost of visiting c. Furthermore, we also account for the subsequent expected cost of visiting, after c, other configurations deemed also potentially interesting, that is, we use a long-sighted, budget-aware approach. Looking-ahead d explorations in the future allows for an estimation of the evolution of the model, which depends on the sampled configurations. For example, at a given moment, considering the predictive model M, if configuration $C1$ is explored, it will lead to the model M_1; while exploring configuration $C2$ leads to the model M_2. Thus, by simulating the outcome of future explorations, and the resulting updates of the model, the system can decide to explore a configuration that may not be the best at that moment, but that, in the end (or in the near-future), leads to a near-optimal configuration that might otherwise not be explored, e.g., using a greedy policy that always explores the configuration that is currently predicted to be the best by the model.

In order to cope with large search spaces (as it is the case for cloud systems), the methodology employed in our work to predict the candidate set of configurations to be explored in the future is based on recent non-myopic extensions [42, 43] of the Bayesian Optimization (BO) approach [51, 14], which incorporate the notion of look-ahead. The candidate set/path is determined on the basis of the model built so far and of its uncertainty regarding untested configurations. The cardinality of this set represents the look-ahead factor, i.e., the number of future exploration steps that are simulated by exploiting the knowledge currently embodied by the model. After having found the candidate set/path, the first configuration is tried and the model is updated. This might lead to changes in the candidate set which is then recomputed.

1.3 Contributions

The main contribution of this thesis is Lynceus, a novel system for optimizing the choice of configurations for the execution of cloud applications, which takes into consideration in its performance model not only the cost of the exploitation phase, but also, and unlike previous systems, the cost of the exploration phase. The configurations found by Lynceus aim at minimizing the overall cost that the user has to pay while ensuring the user imposed Quality of Service (QoS) restrictions are complied with.

We have also gathered extensive datasets representative of the performance and cost of machine learning jobs deployed over large scale platforms in the Amazon EC2 public cloud. More in detail, the datasets were obtained running the training phase of state-of-the-art Neural Networks (NNs). The NNs were implemented using Google's Tensorflow framework and deployed over cloud platforms spanning a minimum of 8 and a maximum of 112 (virtual) cores, considering four different VM flavours, and three application level parameters (affecting the training process of the NNs). This yields a vast configuration space, with each dataset encompassing a total of 384 configurations, whose exploration requires, globally, 64 hours (worst case scenario) and a cost (at current EC2 rates) of 90 USD, approximately. The datasets have been made publicly available and can represent a valuable asset for future works targeting the problem of self-tuning in the cloud.

Lynceus was evaluated using the aforementioned datasets as well as other publicly available datasets, and compared with CherryPick [2], a state-of-the-art approach based on a (shortsighted) BO technique. Our results show that, at parity of budget, Lynceus consistently finds solutions approximately two times closer to the optimum than CherryPick.

1.4 Research History

This work was develop in the Distributed Systems Group of INESC-ID Lisboa. During this period, I benefited from the valuable contributions of several members of the group. In particular I'm grateful to Diogo Barradas, Manuel Reis and Pedro Joaquim for the help with the construction and acquisition of the datasets and to Professor Luís Rodrigues for the fruitful dicussions and comments regarding Lynceus and all work around it.

A paper that describes part of this work and some results has been published in the "Atas do Décimo Simpósio de Informática, INForum 18".

This work has been partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects with references UID/CEC/50021/2013 and PTDC/EEIS-CR/1743/2014. Also, the extensive evaluation presented in this dissertation was possible due to the grant received from the AWS Program for Research and Education.

1.5 Structure of the Document

This thesis is structured as follows: Chapter 2 analyses related work on systems for the self-tuning of cloud applications and on techniques for optimization of expensive objective functions; Chapter 3 describes Lynceus, detailing its main features, design goals and challenges; Chapter 4 presents the results

of the experimental evaluation and finally Chapter 5 concludes this document with some observations and remarks as well as with possible future research directions to further improve and extend Lynceus.

2 Related Work

This chapter begins by providing, in Section 2.1, an overview of the alternatives made available by modern cloud providers in terms of both diverse platforms, as well as costs and pricing schemes. Based on this analysis, we motivate the need for the current state-of-the-art systems that analyze the large space of machines in order to provide users with the best configurations for the deployment of their applications. Section 2.2 overviews some of the most relevant modelling and optimization methodologies that are used by recent systems for self-tuning of cloud applications. Concrete systems that make use of these methodologies are then reviewed in Section 2.3.

2.1 Pricing Models for Resource Provisioning in the Cloud

This section provides an overview of the large offer of Virtual Machine (VM) types and pricing schemes made available by modern cloud providers. The following data considers, in particular, three of the main cloud providers: Google, Amazon and Microsoft, since these providers are the most targeted by the reviewed systems [66, 2, 69, 21, 20, 34, 35].

The existence of multiple providers allows for competition and, therefore, for the prices to come down. However, the huge amount of choice makes it difficult for users to know, first of all, which provider they should select and then which instance type(s) and how many instances they should choose in order to obtain a good enough performance for a reasonable price.

To choose a machine for deployment of an application, a user must first reason on which are the most predominant higher level characteristics of the job and that have more influence in the overall performance. Providers usually offer a choice of machines in 5 broad categories: general purpose, compute optimized, memory optimized, accelerated computing or storage optimized machines. Depending on the providers, some differences may be spotted, for instance, Google has shared-core machine types and doesn't have specific types for accelerated computing.

After choosing the broad category that best fits his job, the user will have to analyze all families that exist in that category, e.g., instances of type 'F1', for Amazon's accelerated computing machine types. Each family has certain specific categories, such as the processors of the machines and the optimizations they provide, for example in terms of network bandwidth or support for enhanced networking.

Once the family with the characteristics that best fit the job has been discovered, the user is required to choose the size of the machine that he wants. Sizes vary between 1 to 96 virtual CPUs (vCPUs) and, typically, a higher vCPU count translates into an increase in available memory.

Usually, all these characteristics are fixed, however some providers, such as Google, allow users to customize their machines [32]. Although there are some rules for the number of vCPUs of each processor and for the amount of memory that can be chosen, the user is still left with a huge amount of

	Amazon	Google	Microsoft
Use Cases	5	5	6
Instance Family	≥ 3	≥ 1	≥ 1
Sizes	≥ 2	≥ 1	≥ 3
Customized	—	3080	—
TOTAL	83	25 (3105)	133

Table 2.1: Amount of virtual machines of each cloud provider

combinations that, if correctly chosen, provide configurations that give optimal performance. Nonetheless, being able to choose wisely is a key factor which, given the range of available possibilities, may be a hard task even for an expert, let alone for a user recently introduced to the cloud environment.

Table 2.1 shows the number of VMs offered by each provider. Disregarding custom machine types, GCE doesn't have such a broad choice. However, when they are taken into account, there are more than 3000 possible machines one can choose from. This count was made based on the assumption that each vCPU must have the same memory. Both Amazon and Microsoft don't offer the option of customizing machines, yet have a large selection. The values for the instance families and for the sizes show the lowest value existent for one case. For example in Amazon's case, there is at least one use case with 3 instance families and several use cases with more. For the sizes, the reasoning is the same: at least one family has 2 sizes available while others have more. All the machines add up to a total showed in the homonymous row.

To differentiate between machine prices, the first telltaling characteristic is the period for which resources are acquired. A user can buy one of two types of resources: on-demand resources, for short running, unpredictable jobs or reserved resources, for long running and predictable jobs. While most machines are available for both types of resources, some can only be acquired for one of the two types. Regarding on-demand resources, the user pays for what he uses, with no long-term commitment. Billing can be per hour or per second, however there is always a minimum payment of one minute, even if only a few seconds are used. As for reserved resources, these imply a commitment of one or three years and offer some discounts when compared to on-demand prices.

Depending on the providers, some more machine types are available. For instance, Amazon also provides EC2 spot instances. These instances are spare compute capacity in the AWS cloud that users can acquire by specifying bids. A bid corresponds to the price a user is willing to pay for the instance. A machine is acquired when the bid specified by the user is higher than the current market price for that instance. These instances run either until the user stops them or until the spot price exceeds the price that was specified. Another possible way through which the machines may be stopped is if the machines are revoked. This happens when EC2 needs that compute capacity back. However, in such a situation, users get a two minutes warning before being evicted. These machines also offer the option of specifying a duration of up to 6 hours, with hourly increments. In this case, they will run until that time has passed or until the user terminates them. These resources are known as revocable resources and although they are prone to evictions and uncertainty, the cost savings they offer make up for those disadvantages when good provisioning strategies are employed, like in the work of Shivaram et al. [66].

After the type of resource has been chosen, the availability zone, which corresponds to where the machines actually are physically, and the operating system that is chosen also influence the prices.

Table 2.2 shows the ranges of prices for each provider and for the two payment options. These values reflect the prices of the cheapest and most expensive machines, independently of their use case, family, size, operating system and availability zone. The cheapest machine has the cheapest operating

Payment choice \ Providers		Price by hour in $$		
		Amazon	**Google**	**Microsoft**
On-Demand		0.0058 — 92.5760	0.0076 — 12.3620	0.0110 — 94.4900
Reserved	1 year	0.0030 — 79.4750	0.0300 — 6.6500	0.0070 — 75.2600
	3 years	0.0020 — 66.8660	0.0210 — 4.6800	0.0050 — 64.2900

Table 2.2: Range of prices of each cloud provider

system, is in the cheapest availability zone, belongs to the cheapest family and has the smallest size. The same reasoning, but on the other way round, applies for the most expensive machine.

By analyzing both tables it is clear that there is an enormous number of combinations of instances that can offer the same performance but that have different costs. Choosing between use cases, families, sizes and types of resources is a choice between hundreds of machines. This task's difficulty is further increased by the fact that the performance of the job in each configuration is only known after trying it. Therefore, if a non-ideal configuration is chosen, this will only be known *a posteriori*, after renting the machines and trying the workload.

This motivates the need for systems, such as the one we propose in Chapter 3, that are able to predict how many instances and of which model should be acquired, given a specific workload and thresholds for performance and cost.

2.2 Background on Modelling and Optimization Techniques

Current work on self-tuning of complex applications aims to find the best values for tuning specific parameters of those applications, and state-of-the-art work on optimizing resource allocation in the cloud has as its objective the discovery of the best configurations for the deployment of applications in the cloud. These optimizations are application specific. Either of these lines of work requires the use of modelling and optimization techniques that may be interesting and valuable to our work and ergo shall be briefly described in the next sub sections.

In this section, we first describe in detail the Bayesian Optimization (BO) technique (Section 2.2.1.1) and the modelling technique Bagging Ensemble of Decision Trees (Section 2.2.2.1), which were employed in our work. Then, the remaining sections refer to relevant optimization and modelling techniques exploited by state-of-the-art solutions for cloud optimization, but which are only briefly overviewed since they do not feature in our solution.

2.2.1 Optimization Techniques

In the following sections we describe in detail the Bayesian Optimization (BO) technique, which is used in our work, and briefly overview the relevant optimization technique of Optimal Experimental Design (OED), which is utilized by state-of-the-art systems.

2.2.1.1 Bayesian Optimization

Bayesian Optimization (BO) [14, 51, 59] is a model based method for finding the optimum value for expensive black-box functions. It is especially efficient in situations where the closed-form expression

of the function being evaluated is unknown, but wherein samples can be extracted at certain points. Furthermore, this technique not only transforms complex problems into a series of simpler ones but also converges to the optimum with a small number of explorations, while also trying to minimize it. Much of BO's efficiency stems from the ability to incorporate previous knowledge about the problem to direct the search process.

This technique rests on the famous "Bayes Theorem", hence the name. Simply put, this theorem defines the probability of an event, given prior knowledge of conditions that might be related to said event. The prior represents the space of possible objective functions, that is, several options for the function that is being optimized. During the optimization process, samples are collected and utilized to update the prior, therefore forming the posterior, which represents the updated beliefs regarding the unknown objective function. An acquisition function guides the search process and the most common of these will be introduced later on.

Sequential Model Based Optimization. Sequential Model Based Optimization (SMBO) [36] is a particular case of BO, which leverages a performance model to guide the search for the optimal configurations/parameters. It is a sequential and iterative process through which models are fitted, samples are collected according to the models' predictions and the models are then updated with the real values obtained through the experiments. SMBO behaves in the following way: *(i)* evaluate the target function f at n initial points and build a training set S with the resulting $\{x_i, f(x_i)\}$ pairs, where x_i are the evaluated points and $f(x_i)$ is the observed performance at x_i; *(ii)* fit a probabilistic model M over S; *(iii)* use an acquisition function to determine the next point x_m to sample; *(iv)* evaluate f at x_m and update the model M with the observed performance; *(v)* repeat steps *(ii)* to *(iv)* until a stopping criteria is met, for example, until the improvement over the current best obtained by further explorations is below some tunable limit. This is a sequential process, as the name states, which, in some situations, and in contrast with parallel optimization techniques, may slow down the search for the optimum. Moreover, fitting an inaccurate probabilistic model over the initial samples can be disastrous, since it introduces an enormous overhead of sampling until the model is restored and improved.

Gaussian Distribution. In most practical applications of the SMBO methodology [24, 7], it is assumed that the performance predictions follow a Gaussian distribution. A Gaussian (Normal) distribution is a continuous probability distribution which is used to represent real-valued random variables whose distributions are not known. The Normal distribution is defined by a mean value μ and a standard deviation k: $f(x) \sim \mathcal{N}(\mu, k)$. This assumption allows for tractable Bayesian modelling of functions and for the derivation of closed-form solutions for many problems.

Common Acquisition Functions. The models fitted over the samples provide relevant information regarding the untested configurations, raising the question of whether to *explore* highly uncertain configurations, i.e., configurations where the real observation has a great deviation from the predicted observation, or to *exploit* configurations that are predicted to have a high quality expected value. The trade-off between these types of observations is balanced by the acquisition function that is utilized to select the next point to sample. There are two common improvement based acquisition functions: the Probability of Improvement (PI) [41] and the Expected Improvement (EI) [50, 38].

PI aims to maximize the probability of improvement (Equation 2.1), that is, to find the point x which has the highest probability of being better than the current best, x^*, i.e.,

$$PI(x) = P(f(x) \geq f(x^*)) = \Phi\left(\frac{\mu(x) - f(x^*)}{\sigma(x)}\right) \tag{2.1}$$

where $\Phi(\cdot)$ is the normal cumulative distribution function, under the assumption that the model's predictions follow a Gaussian distribution with parameters $(\mu(x), \sigma(x))$. PI has the drawback of only exploiting, that is, PI discards points that do not have a high probability of being better than the current best, independently of the uncertainty between their predicted and real values. Hence, PI falls prey to the problem of solely sampling points which are certain to offer improvements, even if only marginal, instead of gambling and trying a point which is more uncertain. A more uncertain point has a lower probability of improving over the current best, but has the advantage of improving more over the current best, when the real value is similar to the predicted one. There are variations of PI which consider a tunable trade-off parameter ξ [65, 37] to balance exploration and exploitation.

EI, on the other hand, takes into account the magnitude of the improvement along with the probability of improvement, thus balancing out the exploration/exploitation trade-off. The improvement of a new point is defined as $I(x) = max\{0, f_{t+1}(x) - f(x^*)\}$ and can be computed due to the predictions of the performance model fitted over past observations. The EI acquisition function (Equation 2.2) selects as the next point to sample the one that is expected to give the maximum positive improvement, that is

$$EI(I) = \mathbb{E}[I(x)] = \int_{I=0}^{I=\infty} (f(x) - f(x^*)) P_M(x|x^*) \tag{2.2}$$

where $P_M(x|x^*)$ represents the density function of the probability of improvement I on a normal posterior distribution characterized by mean $\mu(x)$ and variance $\sigma^2(x)$. Therefore, and by assuming that the performance predictions follow a Normal distribution, the EI can be computed in closed-form according to Equation 2.3

$$EI(x) = \begin{cases} (\mu(x) - f(x^*))\Phi(Z) + \sigma(x)\phi(Z) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases} \tag{2.3}$$

where $Z = \frac{\mu(x) - f(x^*)}{\sigma(x)}$ and $\phi(\cdot)$ and $\Phi(\cdot)$ denote the Probability Distribution Function (PDF) and Comulative Distribution Function (CDF) of the standard normal distribution, respectively. A particular case of the EI is the constrained Expected Improvement (EIc) [29], which is particularly useful in situations where there are constraints to the optimization problem, such as when one wishes to minimize the deployment cost of a cloud application subject to additional QoS constraints. In this case, the EIc can be computed as shown in Equation 2.4.

$$EIc(x) = EI(x) \times P(x \; complies \; with \; the \; constraint) \tag{2.4}$$

Besides these acquisition functions, there are also some others based on confidence intervals (Equation 2.5) [19, 4], namely Lower Confidence Bound (LCB) and Upper Confidence Bound (UCB), which consider the negative and positive maximum deviations from the mean, respectively. The UCB algorithm has several different forms, due to the various possible distribution assumptions on the noise. It is based on the principle of Optimism in the Face of Uncertainty, that is, the actions are chosen as if the environment is as as nice as plausibly possible.

$$\begin{cases} LCB(x) = \mu(x) - k\sigma(x) & \text{with } k \geq 0 \\ UCB(x) = \mu(x) + k\sigma(x) & \text{with } k \geq 0 \end{cases} \tag{2.5}$$

However, all aforementioned acquisition functions are greedy and thus consider only one step look-ahead, which can lead to sub-optimal choices in the long term, as discussed next.

Bayesian Optimization with Look-ahead. Recently, researchers have proposed some new algorithms [42, 43] for the purpose of using BO in fixed budget settings (with respect to the number of explorations) so as to minimize some overall cost. These proposals improve and extend the base BO method by attempting to leverage those greedy acquisition functions while combining them with look-ahead heuristics so as to move towards a long-term reward. More precisely, BO is formulated as a Dynamic Programming (DP) [5] problem. In order to do so, these approaches are based on three common modules: a statistical model, used to represent the unknown objective function; a policy that encodes how the model is updated when new information is gathered; a goal/final quantifiable reward which defines the improvements attained by sampling a given point or ending at a certain state. For instance, a reward function can be defined as the maximization of the acquisition function. However, DP problems have some limitations. The problem of state explosion, which stems from the need to consider all possible subsets of cardinality "look-ahead" at each iteration, leads to the problems of nested expectations and maximizations. The DP recursive algorithm (Equation 2.6), which is used to compute the optimal reward R^* to go, works backwards, i.e., from the end state N up to the initial state n. As can be observed in Equation 2.6, to use the algorithm one needs to compute the chained maximizations and expectations for which there are no known closed-form expressions.

$$\begin{aligned} R_N^* &= max\ \mathbb{E}\big[\text{reward}_N(\text{state}_N, \text{sample}_N, \text{result}_N)\big] \\ R_k &= max\ \mathbb{E}\big[\text{reward}_k(\text{state}_k, \text{sample}_k, \text{result}_k) + R_{k+1}(\text{state}_k, \text{sample}_{k+1}, \text{result}_{k+1})\big] \end{aligned} \tag{2.6}$$

Interesting approaches proposed by recent work [42, 43] use the rollout technique [8, 53] to tackle these problems. Rollout is a look-ahead algorithm that approximates the optimal reward to go by selectively cutting off regions of the search space using a heuristic that avoids the prohibitive costs that would otherwise be incurred by a provably optimal, exhaustive search strategy. Nonetheless, although the nested maximizations problem is solved with this algorithm, there are still nested expectations. These are approximated using Gauss-Hermite (G-H) quadrature [46]. This is a type of Gaussian quadrature used to approximate integrals of the shape $\int_{-\infty}^{\infty} e^{-x^2} f(x)dx$ where N points are sampled, each with a distinct weight w_i and, therefore, we get $\int_{-\infty}^{\infty} e^{-x^2} f(x)dx \approx \sum_{i=1}^{N} w_i f(x_i)$. In this work, we adapt these approaches in order to reduce the cost of the exploration phase.

Stopping Criteria. Usually, and independently of the acquisition function, the exploration ends and the optimum is considered to have been found when the improvement that can be achieved by sampling new points is smaller than some given threshold, like it is done in CherryPick [2]. There are also further stopping conditions, such as stopping upon unavailability of budget in fixed budget settings, like Lynceus, or, for instance, when the EI decreased in the last 2 iterations, the EI for the k-th exploration was marginal and the relative performance improvement in the $k-1$-th iteration did not exceed some threshold [24].

2.2.1.2 Optimal Experimental Design

Another relevant methodology for self-tuning systems, e.g., Ernest [66], is OED [63, 61]. OED establishes a methodology to determine which experimental runs should be performed to estimate a statistical model in an optimal way. Optimality is defined with respect to some statistical criterion and to a given statistical model. Roughly speaking, an optimal experiment design requires a smaller number of experimental runs to estimate the parameters with the same precision as a non-optimal design, hence

reducing the costs of experimenting. OED also works well in constrained design spaces, nonetheless, it is not easy to generalize or to use with non-linear settings, which may be the case when predicting costs of configurations in the cloud.

Experimental designs are evaluated through the use of statistical models. By computing the variance of an estimator we know by how much it is deviated from its mean value. Through the minimization of its variance we get a more accurate estimator. According to the estimation theory for statistical models with one real parameter, the reciprocal of the variance of an estimator is the Fisher's Information [27] of that estimator. The Fisher's Information is a way of measuring the amount of information that an observable random variable X carries about an unknown parameter θ of a distribution that models X. Due to this reciprocity, minimizing the variance corresponds to maximizing the information. When the statistical model has several parameters the mean of the estimator is a vector and its variance becomes a matrix.

Therefore, it is necessary to evaluate the covariance matrix of the estimator in order to choose amongst several possible designs, select the optimal one and build a better model. There are several optimality criterions, a popular one, for instance, is D-optimality [17], which seeks to minimize the determinant of the covariance matrix.

2.2.2 Modelling Techniques

This section describes briefly the most relevant modelling techniques employed by state-of-the-art systems and provides a more detailed explanation on the modelling technique employed for our system's performance model.

2.2.2.1 Decision Trees

A decision tree [11] is a decision-support tool that uses a tree-like graph/structure to make predictions about or to classify events according to their likelihood and to previously gathered knowledge from past experiments. A tree is composed of three types of nodes: decision nodes, that correspond to fixed characteristics that determine each tree path; chance nodes, which are nodes associated with undefined events that have some probability; end nodes, that correspond to the final possible predictions. Decision trees allow to go from observations, represented in the branches, to final conclusions regarding the point's target value and enable the construction of models for classification (deciding to which category an input belongs) and regression (predicting the value of an input based on previous learnt values).

Each tree is grown in the following way: given the set of features and their values, decide which feature is the most representative and apply it as the root of the tree; perform this step recursively, until there are no more features left. The quality of each split is measured by the amount that each feature improves the overall model performance, weighted by the number of observations the node is responsible for: typical performance measures [49, 55] are the Gini Index or the Information Gain.

The Gini Index [11] measures the impurity of a set. A high value means the set is impure and contains a plethora of different elements; a low value means the set is pure and possesses a majority of elements of the same type. The best split is one that gives a subset with a very low Gini Index and the other with quite a high index. The one with the high index will be iterated upon to perform the upcoming splits.

The Information Gain [54] assesses the quantity of knowledge that is extracted from a particular feature regarding the class or the final model prediction, that is, the contribution of that feature for the

final output of the ensemble. Decision trees aim to maximize the Information Gain in order to achieve the best splits and therefore higher model accuracy.

The use of a single decision tree to predict the value or the class of a given input may render the learning method prone to overfitting issues, which leads to an inaccurate and imprecise model. In order to deal with this issue, ensemble methods that resort to several trees for prediction have been developed.

Random Forests. A common technique proposed to deal with overfitting issues and that can be applied to a plethora of Machine Learning (ML) learners is the Bagging Ensemble method [10]. It is a collection of weak learners, trained over different subsets of the training set, and whose outputs are reconciled through some form of voting or averaging. The most voted of these is the final prediction of the ensemble.

A drawback of the general bagging algorithm is that the learners may be too correlated, since, although they get different samples for training, the features considered when building them are the same. This leads to the initial problem of overfitting of the decision trees: having the same tree repeated n times will give the same output as having only that one tree.

The Random Forest [12] technique is conceptually similar to the Bagging Ensemble method. It is applied to the specific case of decision trees and, just like the ensembles, is capable of doing both regression and classification. This technique was developed to reduce the correlation between the learners, that in this particular case are trees. It differs from the general bagging algorithm in that the split point is chosen between a small subset of randomly picked features. This technique is called feature bagging. This way, the trees will observe different features when deciding the splitting point and, although they will always aim for the best point, it varies from tree to tree according to the selected subset. Although decision trees are fast and easy to train, having a single decision tree may damage the performance model. Its predictions are accepted without question because there is no way of confirming their accuracy, i.e, they provide no guarantees on uncertainty. This is a drawback of decision trees which can be reduced through the use of ensembles. With a group of trees, only one amongst several predictions is outputted and therefore those further from a consensus are discredited. This leads to more accurate predictions.

2.2.2.2 Gaussian Processes

A Gaussian Process (GP) [68, 56] is a stochastic process, that is, a set of random variables sampled over time, that follows a multivariate Gaussian distribution. This distribution is a generalization of the one-dimensional normal distribution to higher dimensions. While a Gaussian distribution is a distribution over a random variable, a GP is a distribution over functions that has mean function m, usually set to 0, and a covariance function K.

GPs are often used as a prior function for Bayesian Optimization (BO). After sampling a search space, GPs are used to fit the sampled points in order to create a model. Each sampled point is associated with a normal distribution and the group of all the distributions that are fitted to all the data points makes the GP, which, in turn, creates the model. Predictions of the best next points to sample are then made with the help of the uncertainty estimations provided by the model. Although the predictions of a GP model follow a Gaussian distribution by nature, which obliterates the need to make this assumption when applying such a model to the BO setting, GP models are tricky to implement [59]: not only are the GP parameters hard to tune, but selecting the right covariance function (critical for good performance) is also not trivial. Moreover, dealing with discrete features, as in the case of cloud (e.g., type of VMs),

introduces additional challenges with GPs, whose original formulation assumes that both the output and input variables are continuous [28].

Current systems, such as CherryPick [2] and iTuned [25], use this technique to estimate a model for predicting the best points to sample next.

2.2.2.3 Recommender Systems and Collaborative Filtering

Recommender System (RS) are systems designed to provide users with recommendations on items they may like [26, 57]. In the past years, the use of such systems has increased, namely being used by applications such as Amazon [45], Netflix [6, 30], Twitter [40] or Spotify. In order to know which items to recommend, the system uses an algorithm, for example Content-Based Filtering or Collaborative Filtering (CF) [60]. In this thesis, CF will be briefly explained since it is a technique used by several state-of-the-art systems in the area of self-tuning for the cloud, like ProteusTM [24] and Quasar [21].

CF works by finding a set of users that are similar in terms of their past ratings to the user whom we want to recommend something to. After these users have been found, their ratings for the items are analyzed and the ones that have high ratings and that haven't been rated by the target user, are recommended to him.

The similarity between users can be computed in several ways. The most common two are the Euclidean Distance Score and the Pearson Correlation Coefficient. The former corresponds to the length of the line segment that connects two points. As an example, if we consider the ratings of two film series, for instance Star Wars and Lord of the Rings, then each point would correspond to the ratings that a certain user had given to both those movies. The closer two of those points are, the more similar those users are likely to be. In order to be able to make comparisons, the distances must be normalized. Alternatively, the Pearson Correlation Coefficient takes the common ranked items between two users into account and considers the correlation between those data sets. The correlation can vary between $\left[-1, 1\right]$, where 1 is total linear correlation, 0 is no correlation and -1 is total negative correlation.

RS systems were shown to be very effective with very sparse info, making them a valuable asset for all sorts of applications. However not only do they require offline availability of the gathered dataset (how different applications perform on several configurations), which can be expensive/time consuming to acquire. RS assume also to operate on normalized data, which is not easy to obtain in some scenarios. For instance in the context of com. sys. optimization problems, where we don't have an *a priori* idea of what are the min - max performances of diverse applications (corresponding to users) in different configurations (corresponding to movies) [24].

2.3 Self-tuning of Complex Systems

Broadly speaking there are two classes of self-tuning systems that are worth discussing: systems for optimizing application specific parameters and systems for optimizing resource allocation in the cloud. The former search for the best values for tuning parameters that are specific to those applications, for example, the best values for the learning rate in a neural network training job. The latter focus on the virtual resources that are needed so as to achieve higher QoS, for instance lower application running time, at a lower cost. This section introduces several such state-of-the-art systems, starting with systems

for optimizing application specific parameters in Section 2.3.1 and later discussing multiple systems for optimizing resource allocation in the cloud in Section 2.3.2.

2.3.1 Tuning Application Specific Parameters

Analyzing state-of-the-art systems for tuning application specific parameters is relevant for this thesis not only due the techniques employed in the optimization process, but also since the configurations considered this work encompass application specific parameters as well as cloud parameters.

iTuned. The first system that will be analyzed is iTuned [25]. The goal of this system is to automatically find the best settings for database parameters. iTuned is composed of a planner, that determines the next best experiments to run, and an executor that conducts the experiments.

iTuned's planner chooses the experiments based on a technique called Adaptive Sampling. This technique is very similar to the Bayesian Optimization (BO) technique described in Section 2.2.1.1. In order to get the first samples, Latin Hyper-Cube Sampling (LHS) [62, 47] is used. This sampling technique selects m samples from each of m sub-domains of a parameter. These sub-domains are generated by partitioning the domain of a parameter into m equal sub-domains.

To pick the next experiment, iTuned starts by building a Gaussian Process (GP) model (c.f. Section 2.2.2.2) on top of the initial samples. Then, the Expected Improvement (EI) is computed for the next possible points to experiment. The point to experiment next is the one that exhibits the highest EI. This process ends either when the user is satisfied with the improvements or when the improvement offered by more explorations is below a certain threshold. Because the GP model may be flawed, a cross-validation technique, like the one used by Ernest [66], is used to check how trustworthy the model is. Therefore, the EI threshold is only utilized to stop the search when the cross-validation verification guarantees that the model is sufficiently good.

The executor runs experiments either on resources specified by the user for that effect or on underutilized resources of the database so as not to harm the production workload. To determine whether the resources are underutilized or not, the following policy is employed *"if the CPU, memory, and disk utilization of the resource for its home use is below* 10% *(threshold t1) for the past* 10 *minutes (threshold t2), then the resource can be used for experiments"*. iTuned's efficiency can be improved by adding some features such as eliminating parameters that do not have a significant impact on the performance, focusing more on those which are deemed critical, and using several resources in parallel for running experiments. A drawback of this system is that the exploration cost is not taken into account.

ProteusTM. Another system for self-tuning of applications, that aims to tune Transactional Memory (TM) implementations for specific workloads and that, just like iTuned [25] doesn't consider exploration cost, is ProteusTM [24]. This system is composed of two components: PolyTM, which is a TM library that dynamically adjusts itself and changes TM implementations according to current requirements and RecTM, that is in charge of finding the optimal TM configuration for a specific workload.

PolyTM has a large variety of TM implementations. Having concurrent transactions executing different TM implementations is usually not safe, therefore, for transactions to safely change the TM implementation they are using, PolyTM defines a policy which prevents two different transactions from executing two different TM implementations in parallel.

RecTM has three elements: a recommender, that rates unexplored target configurations according to similar explored configurations; a controller, that chooses the configurations to be tried and notifies

PolyTM to make changes accordingly; a monitor, which controls the quality of the explored configurations and that informs the controller of workload changes so as to initiate new optimization phases.

The recommender uses Collaborative Filtering (CF), described in Section 2.2.2.3. In order to rate configurations a Key Performance Indicator (KPI) is used. This indicator needs to be normalized because there is no information as to the maximum value it can reach for a given workload. To solve this problem, the authors of ProteusTM [24] developed a new technique called rating distillation. This technique maps KPI values to a known scale so that CF can use them to rate configurations.

The controller uses BO, with EI as acquisition function, to select the next point to sample. The samples are modelled using CF and, to ensure there is a balance between exploration and exploitation, the search stops when:

- the EI is lower than some threshold;
- the performance achieved in the previous exploration was lower than some bound;
- the EI of the best experiments to choose decreased in the previous 2 experiments.

The monitor, on the other hand, gathers KPIs from the current implementations in order to provider some feedback to the controller. By monitoring the KPIs it also detects workload changes. If the performance indicator for the current workload drops suddenly and significantly, the monitor notifies the controller of the need for a change in the TM implementation.

The methodologies used by systems like iTuned [25] and ProteusTM [24] can be applied for tuning the configuration of any system (including cloud based systems). However, they do not take into account specific challenges of cloud systems, which are targeted by systems like CherryPick [2], PARIS [69] or Quasar [21]. These systems will now be described.

2.3.2 Optimizing Resource Allocation in the Cloud

Several systems for optimizing resource allocation try to solve the problem: ”In which instances should a given workload be deployed in order to maximize either performance or resource utilization?”. This problem can be addressed from the perspective of the users, directing the aim at finding the best configurations for deployment of those users' applications, i.e. the cheapest ones and that offer the best performance with respect to some metric, or from the perspective of the cloud providers, trying to achieve an efficient utilization of available resources.

Systems focusing on providers tend to disregard exploration and deployment costs, while focusing on finding which resources are needed for each workload and how the several workloads that are running on the same machine may interfere with each other. An example of such a system is Quasar [21].

Quasar. Quasar tries to maximize the utility of cloud facilities by determining the resources needed for each workload. When a workload is received, data is collected according to four different categories: scale-up (number of resources per server), scale-out (number of servers), heterogeneity (types of servers) and interference.

When the profiling results are ready to be used, CF techniques, as described in Section 2.2.2.3, are used for classification purposes. The system compares the profiling results with the available labels so as to decide how to fully characterize the workload. This characterization is then used to map the workload to the available machines, following a policy of allocating the least amount possible of resources

for a given job. This policy allows the search space to be reduced, since the biggest resources are examined first. If smaller resources were analyzed at the beginning, there would be an enormous number of combinations of those resources that would satisfy the constraints. By starting off with the biggest resources, fewer combinations exist, because fewer resources are needed for the deployment.

As the system uses CF techniques, the more jobs it analyzes, the more it learns and the better it can get over time. Moreover, it tries to maximize resource utilization. However, not only the allocations that it suggests may not be close to optimal when it comes to allocation costs, but also there is no differentiation between reserved and on-demand resources. As a result, to overcome these limitations, researchers suggested HCloud [20]

HCloud. HCloud is a system for determining which jobs should be allocated to on-demand resources versus reserved resources. In such hybrid systems, as two types of resources are used, knowing how many of each should be acquired is a problem. Another problem is deciding when to map jobs to the on-demand resources versus to the reserved resources. To address the former problem, the authors use Quasar's [21] estimations for resource needs and measure a job's sensitivity to interference, by analyzing its quality, in the same way that is done in Tarcil [22]. As for the latter problem, a policy was defined to map jobs between the resources. This policy was built on top of three principles:

- Reserved resources are utilized before on-demand resources;
- Applications that can be deployed on on-demand resources ought not delay the scheduling on interference sensitive jobs;
- The utilization limits of the reserved instances should be adjusted dynamically so as to reduce the queuing.

Therefore, the dynamic policy that was built devises two limits. To begin with, a soft limit, below which all jobs are mapped to reserved resources. When this limit is reached, the system starts to differentiate between interference prone and insensitive jobs. To make this distinction, the target quality the jobs need and the quality of previously obtained on-demand resources are calculated. If the quality of the on-demand instances is higher than the one needed by jobs, then they are mapped on on-demand resources. Otherwise, they are deployed in the reserved resources. The second limit that is defined by the policy is a hard limit which defines when jobs need to be queued before reserved resources become available. The soft limit is adjusted based on the queuing ratio.

Although this system already takes into account both types of resources, it is still geared towards maximizing efficiency and utilization of resources, which is beneficial for providers, while still neglecting allocation and deployment costs for users.

Proteus. Directed towards users are systems like Proteus [34], whose goal is to exploit the availability of transient excess idle resources that providers make available. These resources can be revoked when cloud providers need them back.

Proteus considers as target application domain the so called parameter server framework. This is a popular approach for distributed machine learning jobs, which subdivides available resources in workers and servers. The workers share the current parameter values and the training data is divided between them so that they can execute applications concurrently and adjust model parameter values. The servers keep the current values and the workers interact with them to fetch and update these values.

The architecture of Proteus comprises two main components: AgileML, a reconfigurable parameter server system, and BidBrain, which is responsible for managing resources.

AgileML takes care of promoting and demoting nodes according to their reliability, by working in three stages. Initially, a list of the reliable and transient nodes is received. Transient nodes are those that can be revoked, namely spot instances in the case of Amazon Web Services (AWS) (c.f. Section 2.1), while reliable resources are all those upon which the user has full control, i.e., has access to the resources until he shuts them down.

In stage 1, workers are located in every machine, but only reliable machines host servers. In this way, in the event that transient machines are revoked, there are no information losses because all the data is kept in the reliable machines. However, when the ratio of workers to servers becomes too high, there is a network bottleneck, since the servers cannot deal with all the requests from the worker machines. To prevent this, the system moves to stage 2.

In the second stage, some transient machines become active servers and reliable machines become back-up servers. Now the workers interact with the active servers, that push all the information in bulk to the back-up servers. Once again, recovery from revocations and node losses is made possible because all the information is stored in the back-up servers.

In stage 3, all workers are removed from back-up servers. This stage is necessary because running workers in the same machines as back-up servers was found to cause straggler effects.

BidBrain, Proteus' other component, keeps information about current and historical market prices and is responsible for deciding when new resources should be acquired. Its primary objective is to minimize cost per unit work. To estimate this cost, BidBrain considers the probability of eviction, which is translated into free computing time, since when a resource is revoked, the price paid in the beginning of the billing hour is refunded. BidBrain also considers the amount of expected useful compute time, that depends on overheads of evictions and additions. BidBrain makes resource allocation decisions periodically (every two minutes) and a few minutes before the end of each billing hour. Each time a decision is pending, a set of instances is analyzed and if an instance lowers the overall expected cost of the set, it is acquired.

Although this work is a good effort towards aiding users in their choices, it also exhibits some drawbacks. It strives to minimize costs of a specific framework (parameter server), also exploiting spot instance market dynamics. However, it does not tackle the problem of identifying the most efficient platform and application level configurations to be used. For instance, Proteus [34] does not optimize the choice of the irrevocable reliable machines to be used as servers.

Ernest. Another class of user-oriented systems, but that are focused on irrevocable resources, are systems like Ernest [66]. These systems take advantage of the fact that multiple jobs have similar structures in terms of computation and communication, hence allowing performance models to be built based on the behavior of those jobs on small samples of data. In order to instantiate the model, the OED technique described in Section 2.2.1.2 was used to decide which experiments to run, so as to achieve near optimal results, i.e. to build an accurate model, as fast as possible.

To determine how accurate the model is, and because there is not much data available, a cross-validation technique is used. Thus, suppose the model has m training data points. To evaluate the model, the m data points are divided into two subsets: a validation set, with 1 data point; and a training set, with $m-1$ data points. After the division, the training set is used to train the model and the validation set tests it. This process is repeated m times and the results are averaged to produce a final estimation.

The models used by Ernest to characterize applications' performance are specific for analytic jobs and capture only the size of the job (i.e., amount of data to be processed) and the number of machines

used. This limits the applicability of Ernest's approach both in terms of application domains and of configuration parameters that it can optimize. Also, the optimality criterion used by Ernest assumes that exploring any configuration has the same cost, which is clearly not the case in cloud settings.

CherryPick. CherryPick [2] leverages Bayesian Optimization (BO), described in Section 2.2.1.1, to build performance prediction models that are used to predict high quality configurations, i.e., configurations that minimize cloud usage costs and guarantee application performance. At first sight, this may seem like an easy task, however, not only is it difficult to find the right balance between resource prices and the running time of the machines (sometimes the time that is saved by acquiring one more machine may offer more gains than what it costs to get that machine and vice-versa), but also there is a restricted amount of information (each experiment has a cost, hence there are limited runs of cloud configurations that are afforded to search for the model).

The objective function that is optimized by the system minimizes the deployment cost of a given configuration, expressed as a function of the time the machines are up and of how much they cost per hour, with a time constraint to guarantee application performance. GPs, c.f. Section 2.2.2.2, are used as a prior function to model the data points and EIc is used as acquisition function.

The system begins by sampling three initial points using a quasi-random sequence, which will give an estimate of the cost function that is to be minimized. Then, a confidence interval is computed with BO and the next best points to experiment, according to their EIc, are selected. Finally, if the improvement that can be achieved with further exploration is less than some defined threshold and if a sufficient number of configurations have been explored, the best configuration is considered to have been found. If not, this procedure is repeated until those conditions are met.

Although CherryPick [2] minimizes the deployment cost for users and doesn't require a model for each instance type, it has some issues. First, the cost of exploring all configurations until the best is found is not taken into account directly by the model. Also, the search space considered in CherryPick's evaluation is relatively small, consisting of only 66 configurations. Even so, CherryPick requires exploring 6-9 configurations, corresponding to approximately 10% of the whole search space. Considering the data reported in Table 2.1, and the possibility of including in the optimization process additional application level configuration parameters, it is reasonable to expect that systems like CherryPick would require exploring a significantly larger number of configurations in order to discover the optimal configuration in a significantly larger search space. Furthermore, the bootstrapping of the performance model is error prone, since it is done by sampling random configurations. Should these configurations be bad (expensive and/or with low quality), the initial model is either poor, already too expensive or both.

PARIS. A different approach to the same problem is PARIS [69], that also provides performance estimates with minimal data collection but, instead of BO, uses the random forest technique described in Section 2.2.2.1.

Initially, the user provides a representative task of the workload, the desired performance metric and a set of candidate VM types. PARIS then outputs predictions for the costs and performances of the instances provided by the user. To make these predictions, PARIS must have knowledge of the resource requirements of the workload and how it is affected by the different VM types. However, deploying this workload on all machines is too expensive. So PARIS divides the modelling task in two phases.

There is a first offline phase for extensive benchmarking of various workloads with each VM type which only runs one time. The benchmarking collects detailed system performance metrics, fitting broad categories such as CPU utilization, network utilization, disk utilization, memory utilization and system-level features. Each time there are new instance types, the benchmark only has to be rerun on those

new instances. Once all this data has been collected, a series of decision trees are trained for each workload and a forest is built.

The online phase runs the representative task provided by the user on 2 pre-defined reference VMs and collects performance metrics and resource usage information. When the trees have been trained according to the user specified performance metric, the information collected during the profiling phase and a VM configuration are fed to the forest, which outputs the mean and 90^{th} percentile performances. This process is repeated with all candidate VM types. To obtain the cost of choosing those instances, it is assumed that the cost is a function of the performance metric and of the cost per hour of that instance, which is assumed to be known.

Thus, PARIS [69] produces a performance-cost trade-off map that aids users when choosing VM instances. The main drawbacks of PARIS' methodology is that its accuracy is strongly affected by the correct choice of reference configurations and by the representativeness of the data in the training set. Moreover, building an accurate performance model requires more data, which is difficult to collect in this setting due to the expensiveness of the explorations. Once again, the only cost that is considered is the deployment cost, while in terms of exploration nothing is taken into account directly in the model.

Scout. Wishing to tackle CherryPick's [2] and PARIS' [69] drawbacks, researchers proposed Scout [35]. Scout was developed to find configurations that optimized a workload for either least cost or shortest execution time. It gathers the strong features of both CherryPick and PARIS. While from CherryPick it takes the search-based method to accommodate mispredictions and performance variance in the cloud, from PARIS it takes the gathering of historical data (offline benchmarking phase) to understand the preferences of a workload. This way, Scout is not affected by the model bootstrapping problem and the collection of historical data allows it to better infer about workload preferences.

Since building an accurate model requires collecting expensive data, Scout builds a relaxed model to reduce this overhead. Instead of predicting the performance and cost of a given workload in a particular configuration, this model predicts the relative performance of two configurations. Afterwards, to update and improve the model, a search-based method rummages the space of unexplored configurations for promising configurations and better regions. Promising configurations are the ones that have the highest probability of having a lower execution time or deployment cost than the configurations explored so far, whereas searching in a better region speeds up the discovery of a (near-)optimal configuration.

The search process learns from the observations collected along the search path and from performance data of other workloads. This previous data comes from previous runs and is embedded in the historical data. Leveraging this knowledge allows Scout to reduce the number of explorations that are made. To determine different choices for the next step, the search process only requires knowing *"how likely is one choice better than some other(s)"* (relaxed model). Configurations are classified resorting to a technique called pair-wise prediction modelling and placed in discrete classes ("better", "fair", "worse") according to their probability of being better than the best so far (Probability of Improvement (PI)). The search process stops either when it can no longer find a better configuration, i.e, when the probability of improvement over the current best is below a user-defined threshold, or when, due to inaccuracies in the model, it can not find a better configuration.

Scout therefore only optimizes either for least cost or shortest time and tries to minimize the number of explorations. However, exploring less does not imply that the cost of those explorations is lower than if more (and better) explorations were made.

	Technique	Pros.	Cons.
Optimization Techniques	Bayesian Optimization	• Allows balancing exploration and exploitation • Transforms complex problems into a series of simpler ones	• Experiments are run sequentially • Getting the function model wrong can be catastrophic
	Optimal Experimental Design	• Reduces the costs of experimentation • Works well in constrained design spaces	• Not easy to use for general or non-linear models
Modelling Techniques	Decision Trees	• Fast and easy to train	• No information on uncertainty
	Gaussian Processes	• Output is Gaussian by nature	• Not trivial to select the right covariance function, which is crucial for good performance • Hyperparameters are hard to tune
	Recommender Systems	• Shown to work with very sparse information	• Require availability of offline data • Designed to operate over normalized data

Table 2.3: Summary of the pros. and cons. of the reviewed optimization and modelling techniques

2.4 Discussion

This section discusses the main conclusions drawn from the analysis of the related work, both in terms of relevant state-of-the-art systems and in terms of optimization and modelling techniques. With respect to the former, we discuss their missing features and summarize (Table 2.4) their optimization goals so far and the techniques employed. In terms of the latter, we analyze the advantages and shortcomings of each technique, summarized in Table 2.3, to further motivate our choices.

For the purpose of promoting and justifying our selection of optimization and modelling techniques, we present in Table 2.3 a summary of all reviewed techniques together with their advantages and drawbacks. Despite no technique being exempt from drawbacks/limitations, there are some which present more complex challenges and which were less suitable to attain our proposed goals. For instance, regarding the optimization techniques, OED was not suitable since our models were non-linear. On the contrary, BO's disadvantages did not pose such a problem as obtaining an inaccurate model is always a possibility with any technique and running experiments sequentially was not an issue. In terms of modelling techniques, the disadvantage of Decision Trees are easily suppressed through the use of ensembles, unlike the disadvantages of the other two techniques which require not only more testing to tune parameters (for GPs) but also finding a way to normalize data which is unkown *a priori* (in the case of RSs).

The following paragraphs summarize the main features of the reviewed state-of-the-art systems, namely discussing their shortcomings and missing features. To this end, Table 2.4 classifies the systems that were analyzed according to the following dimensions:

- whether they are user-centric or provider-centric;
- their optimization goal;
- the techniques they use to reach the optimization goal;
- whether they take advantage of spot markets;
- whether their model takes into consideration the deployment cost of the configuration;
- whether their model takes into consideration the cost of exploring the configurations.

	Orientation	Optimization Goal	Optimization/ Modelling Technique(s)	Spot Market (Historical Data)	Considers Deployment Cost	Considers Exploration Cost
Proteus	User	Minimize cost x job done at steady state	Time-series based prediction	✓	✓	—
Scout	User	Minimize execution cost or time	Classification techniques (pair-wise prediction)	✓	✓	—
CherryPick	User	Minimize execution cost subject to a performance constraint (time)	Bayesian Optimization	—	✓	—
Paris	User	Minimize generic function of performance metric and VM cost	Random Forest Models	—	✓	—
Ernest	User	Minimize estimation error	Optimal Experimental Design	—	—	—
Quasar	Provider	Maximize resource utilization subject to QoS constraints	Classification Techniques (Collaborative Filtering)	—	—	—
HCloud	Provider	Determine which jobs should be mapped to reserved versus on-demand resources	Uses Quasar's results (classification techniques are implicit)	—	—	—

Table 2.4: Comparison between the state-of-the-art system implementations

Overall, although in the literature a number of useful ideas and powerful modelling techniques have been proposed, none of the existing systems attempt to identify the optimal balance between the cost of exploration and the gains achievable during the exploitation phase.

Most of these systems build prediction models to guide the exploration of the search space. The guided exploration aims to find the optimal configuration faster, i.e., to reduce the number of configurations that are evaluated. However, to explore relevant configurations, the model needs to be precise and accurate and, to build such a model, real data is necessary. Nevertheless, in the cloud setting, gathering data to build accurate performance models is quite expensive, since each time an application is deployed in a configuration, the user incurs a cost. This implies there is a trade-off between exploration and model accuracy.

Furthermore, and although some systems [35] do try to reduce the number of explorations, given the high heterogeneity of the configurations' costs, exploring a smaller number of configurations does not necessarily translate into a cost reduction. For instance, consider a scenario with two different exploration options:

- *Option 1)* explore 10 configurations with cost 1 and 5 configurations with cost 10. This yields a total of 15 explorations with a cost of 60;
- *Option 2)* explore 5 configurations with cost 15 and 5 configurations with cost 1. This yields a total of 10 explorations with a cost of 80.

Option 1, which explores 5 configurations more than option 2, is definitely preferable cost-wise.

Additionally, and although these systems find a near-optimal configuration for deployment of user

specific workloads, and some of them, such as CherryPick [2] and Scout [35], even consider the deployment cost of the configurations in the model, none is concerned with the exploration cost (Table 2.4), which can be quite expensive.

Besides this, existing BO approaches have two limitations. First, they do not take into account the cost of the sampling phase. The acquisition function may suggest to sample very expensive configurations, resulting in a highly expensive initial training phase. Second, they implement a "myopic" search strategy. At each iteration, BO selects the point that maximizes the acquisition function. Thus, this approach falls prey to the same shortcomings of greedy search approaches [41, 50, 19, 4], i.e., getting stuck in local maxima.

Summary

The increasing use of cloud services and the competition between providers led to an increment in the number of machines available for deployment of jobs and applications by the users. The search space of current machines available was shown in Section 2.1, namely in Table 2.1. The availability of such a broad choice of VM configurations, whose selection may have an impact also on the tuning of internal parameters of the application (e.g., the choice of the training policy for a distributed ML platform is typically affected by the scale of the underlying cluster [52]) faces cloud users with a complex optimization problem, motivating the need for automatic solutions.

Furthermore, since the performance of an application on a given configuration is not known *a priori*, finding the optimal configuration requires testing and experimenting several configurations, which is both time consuming and expensive.

Hereupon, researchers developed the current self-tuning systems for optimizing the configuration of complex (cloud) systems that were analyzed in Section 2.3. Although these systems find a near-optimal configuration for deployment of user specific workloads, and some of them, such as CherryPick [2], even consider the deployment cost of the configurations in the model, none is concerned with the exploration cost, which can be quite expensive.

3 Lynceus

To tackle the limitations identified in Chapter 2 and to fill the gaps left by previous self-tuning systems, this chapter introduces Lynceus, a system for self-tuning of cloud configurations for deployment of user-specific applications. The name Lynceus comes from the homonimous Argonaut who served as lookout on the Argo and who was said to have excellent eye-sight, enabling him to see through walls, trees, skin and the ground.

Previous systems were focused only on maximizing the quality of the final/steady state configuration, without keeping into account the cost dynamics associated with the exploration of applications' and platforms' configurations in the cloud. Lynceus, instead, is a long-sighted, budget-aware approach that aims to strike an optimal balance between the cost associated with the exploration phase and the quality/cost of the final configuration identified by the self-tuning process. This is achieved by selecting as the next configuration to explore the one that maximizes the efficacy and efficiency of the *overall* sampling phase, without overspending the available budget B. To this end, at each iteration, Lynceus speculates not only on the outcome of sampling a single configuration, but on the outcome of sampling several configurations according to different sequences of future explorations (which we call exploration paths). The efficacy of this approach stems from the fact that instead of simply pursuing the identification of the best configuration at each iteration (like greedy strategies do), it finds the configuration x that exhibits the best (tunable) trade-off between cost at runtime and exploration cost.

The remainder of this chapter is devoted to describing:

- the design challenges we had to overcome during the process of implementing Lynceus;
- the several components of Lynceus' architecture, specifying their implementation details;
- the missing features and future system improvements and optimizations.

3.1 Challenges

Similarly to state-of-the-art systems [2, 35, 24], Lynceus builds a performance model to guide the exploration of the search space. This model is iteratively updated and improved through the exploration of new and unexplored configurations. Previous systems (e.g. CherryPick [2] and Ernest [66]) strive to minimize the number of explored configurations, neglecting the heterogeneity of their costs. Conversely, by keeping exploration budget and exploration cost explicitly into account, Lynceus plans the exploration phase with the ultimate goal of maximizing the quality of the identified solution given the budget actually available for exploration. Below, the main challenges encountered while building this system are reported:

- Deciding how to select the configurations to explore in a way that would not only allow for the most

explorations but also that assured that each exploration was the most beneficial for the model at that moment;

- Creating datasets for training, testing and evaluating the system. There are no well known and standard benchmarks publicly available and not only does building these datasets cost money, but also poses quite a challenge when it comes to selecting the most relevant application specific parameters to optimize. The goal when building these datasets was selecting the most representative and influential parameters for the application's performance;
- Creating realistic heuristics for an early timeout policy, so as to stop the exploration as soon as the configurations are deemed unworthy, either for being too expensive or for not complying with the user imposed constraints.

Throughout the following sections, we discuss in detail the approaches undertaken while developing the Lynceus system. We start by providing an overview of Lynecus in Section 3.2. Section 3.3 describes the core optimization algorithm of Lynceus. Finally, Section 3.4, concludes this chapter with a set of relevant optimizations to improve Lynceus' performance.

3.2 System Overview

Lynceus is a system for optimizing the choice of configurations to execute cloud applications. Lynceus explores the space of configurations in search for the optimal one, which is defined as the one with least global cost (exploration cost + exploitation cost) and that complies with the user-defined QoS constraints, under two additional type of constraints: *(i)* a constraint defining the available budget to be used for the exploration phase; *(ii)* user-specified constraints on the QoS levels that should be satisfied by the chosen configuration (e.g., the maximum training time of a machine learning model should be below 10 minutes and ensure at least 85% accuracy).

Lynceus resorts to an iterative process to build, update and improve a performance model so that in the end an optimal configuration is selected and outputted to the user. Abstractly, it can be thought of as a sequence of blocks that interact with each other and with the external provider, as illustrated by Figure 3.1. At each moment/iteration, Lynceus is characterized by a state. This state is composed by the available budget for experiments, two sets of configurations (the explored and the unexplored sets) and the last configuration explored. Each time there is a new experiment, the state is updated.

Each block in Figure 3.1 represents a phase of the self-tuning/optimization process:

- **Sampler:** executes upon start-up. Selects *k* random configurations which are then sampled and executed to collect initial performance and cost measurements. These measurements constitute the initial training set and act as baselines upon which the performance model is built. The selection method of these configurations, in the current prototype, is a uniform random strategy, for simplicity, although it would be relatively straightforward to incorporate alternative initial sampling policies, e.g., Latin Hyper-Cube Sampling (LHS) [62, 47], that guarantees a more uniform initial selection of configurations among the search space;
- **Modeler:** consists of the performance model which is built on the basis of the initial samples gathered by the Sampler. The model is then updated and improved iteratively, based on the information received regarding the new explorations' performance and cost data. This continuous

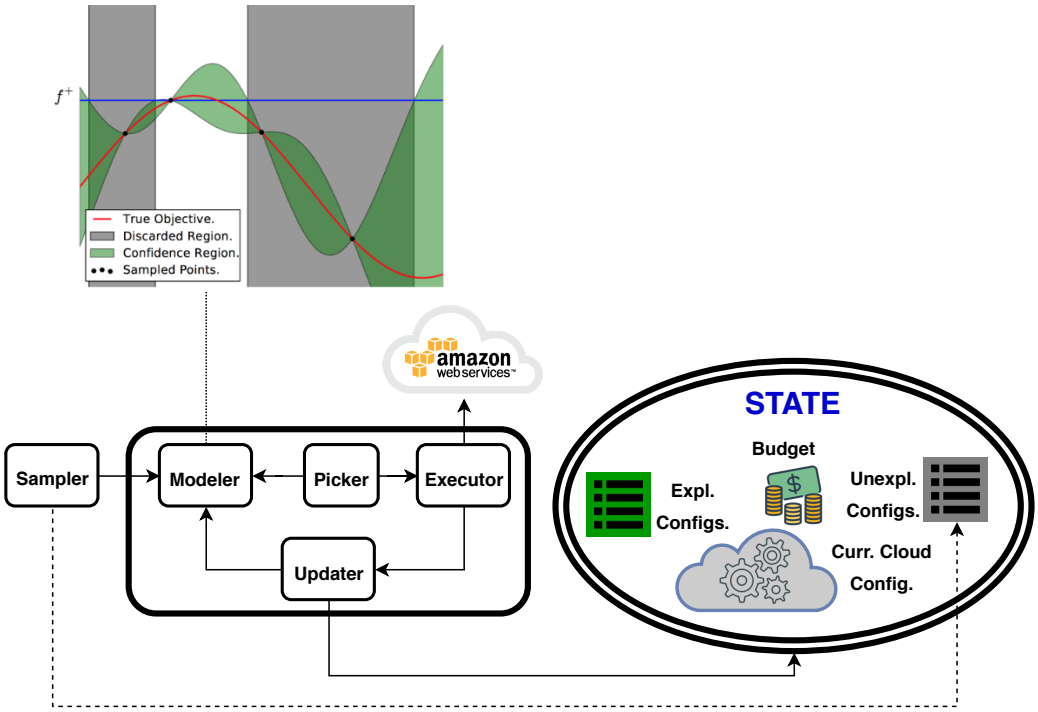


Figure 3.1: Lynceus system overview

process ensures the accuracy of the model and, therefore, more precise predictions concerning the next most beneficial explorations. This guided exploration permits a cheaper exploration phase which takes the most advantage of the available budget;

- **Picker:** incorporates the acquisition function (Section 2.2.1.1) that, by analyzing the performance model, selects the next configurations to explore. The Picker then informs the Executor of the configuration that is to be tested;

- **Executor:** connects to the selected cloud provider and runs the scripts to execute the user specified application on the configuration that the Picker selected for exploration. Upon completion, all the information concerning the experiment is forwarded to the Updater. However, deciding for how long to run the experiment and when to stop it is quite a challenge, since running the job until completion might take too long, and since there is no way to know how long it will take to achieve a given performance threshold;

- **Updater:** after receiving the information regarding the new experiment, it updates: the available budget, subtracting from the current budget the cost of the experiment; the current configuration, which becomes the one that was just explored; the set of explored configurations, adding the current one; the set of unexplored configurations, removing the current one. Then, it informs the Modeler of the performance and cost of running the application on the current configuration.

The following section describes each of Lynceus' components in more detail, fully explaining the algorithm and how the several challenges that arose during its development were dealt with.

Algorithm 1 Lynceus

```
1: function LYNCEUS
   ▷ Sample init configs and update state variables
2:     S ← initS()
3:     while B > 0 ∧ U ≠ ∅ do
4:         (c, U_c) ← NextConfig(S, h)
5:         if (c == null ∨ U_c ≤ ε) then                ▷ Stop exploration
6:             return argmin_cost(c){S}
7:         else                                         ▷ Update model and state
8:             cost(c) ← sample(c)
9:             E ← E ∪ {c, cost(c)}
10:            B ← B − cost(c)
11:            U_unex ← U_unex \ {c}
12:    return argmin_cost(c){S}
```

3.3 The Algorithm

This section fully describes Lynceus. Firstly, in Section 3.3.1, we present an explanation of a simplified version, which is not feasible in a real setting due to the magnitude of the search space. Later on, in Section 3.3.2, we report the full algorithm and the heuristics utilized to render the algorithm useful in practice.

3.3.1 General Description

The overall idea of Lynceus consists of finding the next best configuration to sample based on the current state of the system. Algorithm 1 shows the overall flow of the system. The Sampler is represented by line 2, which corresponds to the initial bootstrapping of the model. Each next best configuration is deployed and the state of the system is updated. Lines 9-11 of Algorithm 1 can be interpreted as the Updater. There are several possible ways for the exploratory process to come to an end: there are no more configurations to explore, which can happen either because all configurations have been explored (this corresponds to a brute force search that in practice never happens) or because none of the unexplored configurations is considered feasible; the improvement attained by further explorations is lower than a tunable threshold; there is no more budget available for further explorations. Whenever neither of these conditions is verified, the iterative search and improvement goes on and the Picker (represented by line 4 in Algorithm 1) comes into action to select the next configuration to explore. Otherwise, the exploration stops and the optimal configuration x^{opt} is considered to have been found.

Assumptions. In order to fulfill the goal of finding the optimal configuration, Lynceus builds a cost model of an application when executed in the cloud in different configurations. This application may be, for instance, a graph analysis process or the training of a model based on machine learning techniques. It is assumed that the job takes an unknown finite time to finish and that this time depends on the chosen configuration, not only in terms of VMs (number, types and sizes), but also of internal application parameters, such as the batch size and the learning rate in the case of machine learning applications. Furthermore, in this work we assume that the cost of executing the job in a given configuration is directly proportional to the time during which the resources (VMs) are allocated and that the costs per unit of time (usually seconds) are known *a priori*. Therefore, by having the model predict the expected cost of a given job in a particular configuration, we can automatically derive an estimate of the time it will take

to run that job until completion or until achieving a good enough accuracy, for example in the case of a neural network training job.

Bootstrapping the Algorithm. The search process is bootstrapped by the Sampler through the exploration of k initial samples, selected randomly from the set of unexplored configurations U_{unex} (a.k.a. training set). These samples are used to build the Modeler, that is, to construct the base cost model for future predictions of the configurations to explore. Building on this ground model, the Picker, along with the Executor and the Updater, execute the refinement process (Algorithm 1, line 3) of the Modeler.

State of the System. The state at each iteration i is defined by the quartet $S_i = \langle B^i, c_i, E^i, U^i_{unex} \rangle$, where B^i is the available budget at that moment to spend on further explorations, c_i is the current configuration, E^i is the set of explored configurations (a.k.a. test set) and U^i_{unex} is the set of unexplored configurations (training set).

Budget Considerations. As previously mentioned, the algorithm is aware of the budget B available at each iteration for further exploration and refinement of the model. Thus, Lynceus knows the exploration process must come to an end when it runs out of budget. At each iteration i, the cost of exploring c_i is deduced from the available budget, $B^i = B^{i-1} - \textit{cost}(c_i)$.

Model, Test and Training Sets. In each iteration, the Executor deploys a configuration in the cloud and collects performance and cost measurements. The Updater then proceeds to update the state of the system, namely adding the configuration to the set of explored configurations E^i (test set), taking it from the training set and subtracting the cost of the exploration from the budget. Each sample consists of a tuple $\langle c_x, \textit{cost}(c_x) \rangle$, where c_x denotes a given configuration and $cost(c_x)$ is the cost of sampling the job on that configuration. This information is sent to the Modeler to update and refine the model.

In practice, consider U^{i-1}_{unex} the set of unexplored configurations (training set) at the end of iteration $i-1$. At iteration i, the algorithm ought to select a configuration $c_i \in U^{i-1}_{unex}$ that, along with the corresponding exploration cost, will be added to the test set, i.e., $E^i = E^{i-1} \cup \langle c_i, \textit{cost}(c_i) \rangle$.

Choice of the Next Configuration. One of the biggest challenges is the selection of the next configuration $c_i \in U^{i-1}_{unex}$ to be explored in each iteration i. Unlike previous work, which leverages greedy search strategies like Bayesian Optimization (BO) [2] and that does not consider the exploration cost, Lynceus is a long-sighted, budget-aware approach for optimizing the selection of configurations for execution of user-specific applications. Therefore, and to look-ahead, the Picker resorts to ideas from recent work on the field of BO with limited budget [43, 42] which are based on simulating the future and creating a path of configurations to explore.

The rationale behind these algorithms is that perhaps it may be better in the long run to explore, at a given moment, a configuration $c2$ which is worse than the best configuration $c1$. The reason for this is that, since the model changes in accordance with the exploration results, i.e., is updated differently depending on the explored configuration, the worse configuration $c2$ can lead to configurations $c2.1$ and $c2.2$ which are better than the configurations $c1.1$ and $c1.2$ that would be explored after $c1$. Thus, selecting c_i should contemplate not only the cost of trying that configuration but also the expected contribution brought by the experiment to improve the quality of the model. When calculating this contribution, the algorithm considers not only the contribution brought by the tuple $\langle c_i, cost(c_i) \rangle$, but also the contribution brought by the following d simulated iterations $i+1$, $i+2$, $\ldots$, $i+d$. In other words, an in-depth estimation of the expected contribution is performed, gauging the effect of the choice up to d configurations in the future. Thus, we denote the expected contribution of a certain exploration path of depth d as $\textit{path\_contrib}(c_i, c_{i+1}, \ldots, c_{i+d})$.

Algorithm 2 Choice of the Next Config

```
1: function NEXT_CONFIG(S,depth)
2:     M ← ERT(S)                          ▷ Train a new Ensemble of Random Trees
3:     V ← {c ∈ U_unex : P(cost(c) ≤ B^i) ≥ β}   ▷ Configs that comply with the budget
4:     if V = ∅ then return (null, 0)      ▷ Stop exploration
5:     else
6:         ∀c ∈ V, (U_c, C_c) = UTILITY(S, c, depth)
7:         return (c, U_c) : argmax_{c∈V}{U_c/C_c}
```

Ideally, in each iteration, the algorithm would compute and explore all the exploration paths of depth d, considering all the unexplored configurations. Following this approach, the number of paths to analyze would be of the order of $|U_{unex}|^d$. For each path, the algorithm would compute the expected contribution of that path to the model. Later on, it would apply to each path a function F that weights the path's contribution with its cost *path_cost*$(c_i, c_{i+1}, \ldots, c_{i+d})$. Finally, the configuration that is chosen is the one that maximizes F and state attributes are updated. However, and taking into consideration the numbers displayed in Table 2.1, the training set U is too wide for an exhaustive exploration of all possible paths of depth d, which would be too expensive, both in terms of time and cost, and would have an infeasible computational complexity. In reality, this approach is not possible, especially considering the computational capacity of most common computers and laptops.

Stopping Conditions. Various stopping criteria can be plugged into Lynceus. Clearly, the exploration concludes whenever the optimization process runs out of budget or explores all the available configurations. In addition, one may use some of the heuristics typically employed in the literature, e.g., stopping the exploration should the model predict that there are no other configurations satisfying the QoS constraints (i.e., the set of feasible solutions, according to the model's predictions, is empty) or if it predicts that none of the unexplored configurations improves over the current best (i.e., reduces cost in our case) by more than a fixed threshold [24, 2].

Selecting the Optimal Configuration. The configuration selected once the search process concludes, i.e., when there is no more budget, is the best configuration (as in cheapest and that complies with the QoS constraints) that belongs to the set of explored configurations E.

3.3.2 Detailed Description

The following paragraphs describe the system in detail, specifying the heuristics, approximations and policies developed and utilized in order to make the algorithm usable in a real world setting. As previously stated, unlike systems [2, 25, 24] based on greedy SMBO, we intended to analyze not only the next configuration but the next d configurations, so as to improve the exploration strategy. The technique proposed to fulfill this goal will be described hereafter.

Selecting the Next Configuration. Following the bootstrapping phase and whenever the Picker is required to select the next configuration to explore, the first task consists of building the prediction model M (Algorithm 2, line 2). Due to the magnitude of the search space, assessing the quality of all unexplored configurations would be extremely inefficient. Therefore, the first heuristic used by the algorithm to avoid this overhead consists of the creation of the feasible set V (Algorithm 2, line 3). This is the set of all the configurations that are analyzed and includes only the feasible ones. A feasible configuration is defined as a configuration that is predicted to comply with the budget, i.e., $V = \{c \in U_{unex} : P(cost(c) \leq B) \geq \beta\}$. Then, for each feasible configuration, Lynceus will simulate the future (Algorithm 2, line 6) and try to

Algorithm 3 Computation of Utility with rollout

1: **function** UTILITY(S, c, depth)
2: $M \leftarrow$ ERT(S) ▷ *Train a new Ensemble of Random Trees (ERT)*
3: $U \leftarrow EI_c(c)$ ▷ *Compute constrained EI*
4: $C \leftarrow cost(c)$
5: **if** depth $= 0$ **then return** (U, C)
6: **else**
7: $(a_j, w_j) \leftarrow GH(f_x), j = 1, \ldots, N$ ▷ *Coefficients of the G-H quadrature*
8: **for** $j = 1, \ldots, N$ **do**
9: $S' \leftarrow \langle c$; $E' \leftarrow E \cup (c, w_j)$; $O' \leftarrow O - w_j$; $U'_{unex} \leftarrow U_{unex} \setminus \{c\}\rangle$
10: POLICY(S')
11: **if** $c' = null$ **then** continue ▷ *There's no suitable x'*
12: $(u, c) \leftarrow$ UTILITY(S', c', depth $- 1$)
13: $U \leftarrow U + \gamma a_j u$; $C \leftarrow C + \gamma a_j c$
14: **return** (U,C)

predict which would be the next configurations to be explored. The amount of simulations that are made depends on how deep in the future we want to look-ahead.

Reducing the Search Complexity. Ideally, Lynceus would explore all look-ahead paths. However, as this is not viable in reality, there are some approximations to reduce the search complexity. Assuming one wanted to perform a search of depth d, the total amount of paths to explore would be $|V|(|V|-1)\ldots(|V|-d)$. For realistic dimensions of the feasible set V, that is, from 50 configurations upwards, the complexity of exploring all paths would be unacceptable. Recent works [43, 29] examine different approaches that aim to reduce this problem's complexity. In this work, we use a heuristic conceptually similar to those proposed. It consists of a first, exhaustive, exploration in breadth of all v feasible configurations in V, followed by in-depth explorations for the remaining $d-1$ steps, so as to avoid an exponential growth of the search process. That is, for the first exploration, all feasible configurations are considered. However, for the remaining depth explorations, only one path per configuration is built. Hence, the algorithm builds merely v paths. The breadth search (Algorithm 2, line 6), associated with the first exploration step, requires these paths' utility to be evaluated.

Depth Search. The depth search, described by Algorithm 3, works recursively. Initially, model M is built and the utility and cost of the current configuration c are calculated. After, the current state S is cloned so as to guarantee that the algorithm is able to return from the recursive calls without corrupting state S due to the in-depth simulations. This way, it is assured that both the test and training sets are always correctly up to date. Posteriorly, the algorithm selects the configuration that is predicted to offer the most improvement when compared with the current optimum (Algorithm 3, line 10) so that its utility is estimated. This estimation requires this process to be repeated until the desired depth d is attained. At this point, the algorithm returns to the initial depth with the path's contribution estimated.

When Lynceus is used with look-ahead 0, there are no simulations and the acquisition function of SMBO coincides with the EIc per dollar [36], that is by how much we expect to reduce the cost of the final configuration, after exploring the current configuration.

Contribution of a Path. A path's contribution corresponds to the sum of all the utilities of the configurations that make it up. The utility of a configuration is its EIc, computed according to Equation 2.4. In this way, we get $path\_contrib(c_i, c_{i+1}, \ldots, c_{i+d}) = U$.

To advance in depth, starting from a configuration of depth $i \geq 1$, it is necessary to estimate the utilities of configurations of depth i', having $i < i' \leq d$. This involves computing nested expectations, for which there is no known closed form expression. As a consequence, these values are approximated

Algorithm 4 Policy function

1: **function** POLICY(S)
2: $M \leftarrow \text{ERT}(S)$ ▷ *Train a new Ensemble of Random Trees (ERT)*
3: $V \leftarrow \{c \in U_{unex} : P(cost(c) \leq O) \geq \beta\}$ ▷ *Configs that comply with the budget*
4: **return** $c : argmax_{c \in U_{unex}}\{EIc(c)\}$

resorting to *Gauss-Hermite* (G-H) quadrature [46], which is a type of Gaussian quadrature used to approximate integrals where N points are sampled, each with a distinct weight. Currently, our algorithm uses 3 points for the approximation. By applying the quadrature to the Gaussian distribution associated with the model's prediction for the current configuration c at a given depth i, N pairs (a_j, w_j) are obtained. w_j corresponds to a possible cost for c and a_j is the weight associated with that cost. In fact, weight a_j is indeed related to the likelihood of the cost being w_j and it is used to determine the contribution of w_j to the estimation of the total utility of the path. Parameter γ (Algorithm 3, line 13) is a discount factor [64] which allows for the calibration of the weight of the predicted utility of a configuration based on the depth level (distance in the future) in which the prediction is estimated. A value of $\gamma = 0$ nullifies the contribution of configurations in a path of depth higher than zero. In contrast, a value of $\gamma = 1$ assigns the same weight to all predicted utilities for all configurations of a path, independently of their depth.

For each estimated cost w_j of the current configuration c, that is, for each of the N points, the algorithm is required to select which would be the next configuration in the path predicted by the model after it had been updated with the "simulated" sample (c, w_j). This process is performed by a policy function described by Algorithm 4. The policy works by first updating the model, then defining the set of feasible configurations (i.e., the ones that have probability β of having a cost lower than the available budget) and, lastly, picking the configuration from this set that maximizes the EIc.

3.4 Early Timeout Policy

This section introduces an optimization aimed at enhancing the effectiveness of Lynceus, namely an early time out policy that stops the exploration of suboptimal configurations.

In a real world setting, a user experimenting configurations is able to automatically classify one configuration as worse or better than the current optimum by observing the application running on each configuration for some limited time interval. To improve our system and make a better use of the available budget, we decided to implement a policy that would mimic this behaviour.

The base algorithm presented so far assumes that whenever a configuration is deployed to run a target cloud application (e.g., the training of a complex ML model), it is executed until completion in order to obtain an accurate estimation of its cost. In this section, we introduce *early timeout* policies that interrupt the exploration of configurations as soon as these are detected to be suboptimal, i.e., when they exceed the cost of the current best configuration, or unfeasible, when they are detected to violate some user-specified QoS constraint. By timing out as early as possible the exploration of suboptimal/unfeasible configurations, the exploration cost for that configuration gets proportionally reduced, thus enhancing the cost effectiveness of the exploration process. Although apparently quite straightforward, this optimization introduces a non-trivial challenge: by interrupting the exploration of a configuration at an arbitrary point of the execution of a job, one is left with the problem of determining which value to feed to the cost model used by Lynceus to drive the optimization process. We tackle

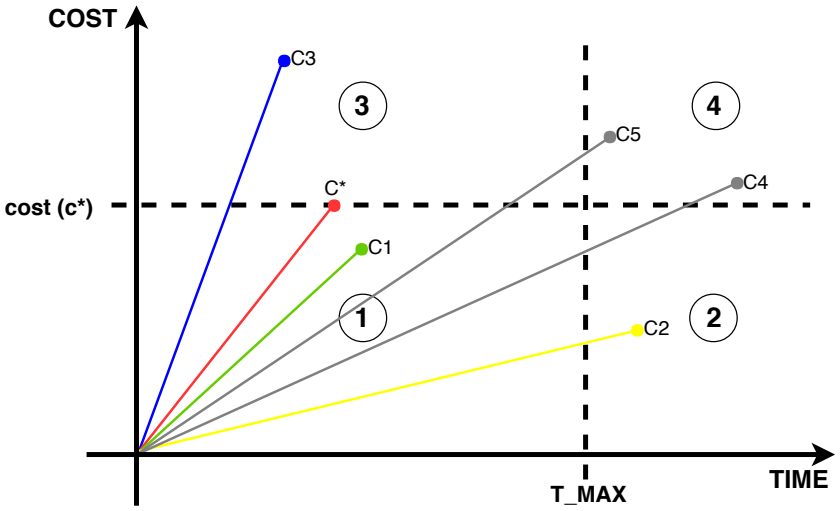


Figure 3.2: Early timeout policy

this problem by integrating a mechanism that, upon the early time out of a configuration, estimates the expected cost of fully sampling the target job based on the progress it achieved upon its time out.

The intuition behind this policy, which is represented in Figure 3.2, is that the first sampled configuration that complies with the user defined constraints regarding the maximum running time *T_MAX* for the job becomes the current optimum $C*$. Henceforth, the exploration of a configuration is interrupted either as soon as the maximum allowed running time is reached (configuration $C2$, second quadrant, Figure 3.2) or as soon as the current configuration achieves a cost as high as the current optimum's cost (configuration $C3$, third quadrant, Figure 3.2). The job runs until completion only when there is no current optimum yet, or when the configuration under exploration is feasible and cheaper than the current optimum (configuration $C1$, first quadrant, Figure 3.2).

Whenever the exploration of a configuration is timed out, as already mentioned, in order to update Lynceus' cost model, it is necessary to predict the expected cost of running the target cloud optimization until completion. To tackle this problem, we monitor the application's progress periodically and detect, at monitoring point, its progress rate (e.g., the percentage of data analyzed in an analytics job or the accuracy reached by a ML model) and whether it violated the specified QoS constraints (e.g., the job execution exceeded some predefined threshold).

If the monitoring system detects an early time out condition, we use the progress data gathered during the sampling of that configuration to estimate its expected completion time and, consequently, its cost (given the time-based charging policy of cloud providers). This can be formulated as a regression problem, which could be addressed with a plethora of alternative black-box models. These, as well as arbitrary learners, can be integrated in the Lynceus prototype. For simplicity, though, the current implementation has been tested with linear models, which, as we will see in Section 4.5, despite their simplicity, result quite effectively in practice.

It should also be noted that, in the current prototype, the monitoring process takes place periodically, with a user-defined frequency. The higher the monitoring rate, the earlier suboptimal configurations can be detected, but also the higher the overhead introduced for monitoring. The latter is strongly application dependent and may vary strongly, e.g.: monitoring the throughput of a database system is way less expensive than evaluating the accuracy of complex ML models using cross-validation of large test sets.

Summary

This chapter presented Lynceus, a novel long-sighted, budget aware system for selecting the optimal configuration for deployment of user specific applications, describing its most prominent features and the challenges that arose during its implementation. Lynceus leverages novel techniques [43, 42] that build on and extend the base Bayesian Optimization (BO) technique in order to be less greedy and to optimize the end reward. Furthermore, and unlike state-of-the-art systems, Lynceus incorporates the notion of budget and aims to reduce not only the cost of the final steady-state exploitation phase, but also of the exploration phase. The next chapter presents an extensive experimental evaluation of the current prototype.

4 Evaluation

This chapter details the experiments performed to test and validate the proposed system. Section 4.1 starts by describing the baselines for comparing Lynceus and detailing the evaluation metrics. Then, Section 4.2 presents and characterizes the datasets used for testing Lynceus on the quality of its final prediction and on its ability to explore the space of configurations. Section 4.3 details the implementation of the system as well as the settings for running the experiments. This section also details some implications of the initial model, which are relevant for understanding the dimensionality of the process of finding the optimal configuration. In Section 4.4 we evaluate Lynceus' ability to find the optimal configuration for the execution of cloud applications according to the specified metrics and we assess the computational complexity of all tested approaches. Finally, Section 4.5 describes the improvements on the quality of the final configuration attained through the use of the timeout policy, comparing both the ideal and the real policies.

4.1 Evaluation Setup

This section describes the selected state-of-the-art approaches with which Lynceus is compared, justifying the reasons for these choices. Furthermore, it presents the metrics used for evaluating all systems in a fair, thorough and systematic way.

Baselines for Comparison. Lynceus is compared with CherryPick [2] not only because CherryPick represents the state-of-the-art when it comes to using BO in the cloud setting, but also because it aims at minimizing the final cost for the user and because it is subject to QoS restrictions imposed by the user. In its original version, CherryPick [2] does not possess the notion of budget. This implies that should the budget be surpassed, the exploration does not stop. Therefore, and for a fairer comparison, we equipped CherryPick with this notion. Lynceus is also compared with a Random approach which selects the next configuration to explore randomly from the set of unexplored configurations. Lynceus is not compared, however, with Scout [35] as there was no implementation available online. We relegate the comparison of both systems to future work, provided that an implementation of Scout is made available by its authors.

Evaluation metrics. To evaluate Lynceus, we employ two metrics: the Distance From Optimum (DFO) and the Number of Explorations (NEX) performed. The former allows us to characterize the quality of the solutions found: a better solution is closer to the optimum, with the best solution C^* having $DFO(C^*) = 0$. The latter enables us to draw conclusions concerning the relationship between amount of search space explored and quality of the solution.

The DFO is obtained by calculating the difference between the cost of the optimal configuration and the cost of the chosen configuration, normalized by the cost of the optimal configuration, i.e., $DFO = \frac{cost(c_{chosen}) - cost(c_{opt})}{cost(c_{opt})}$. This way, the smaller the DFO, the better the final configuration selected.

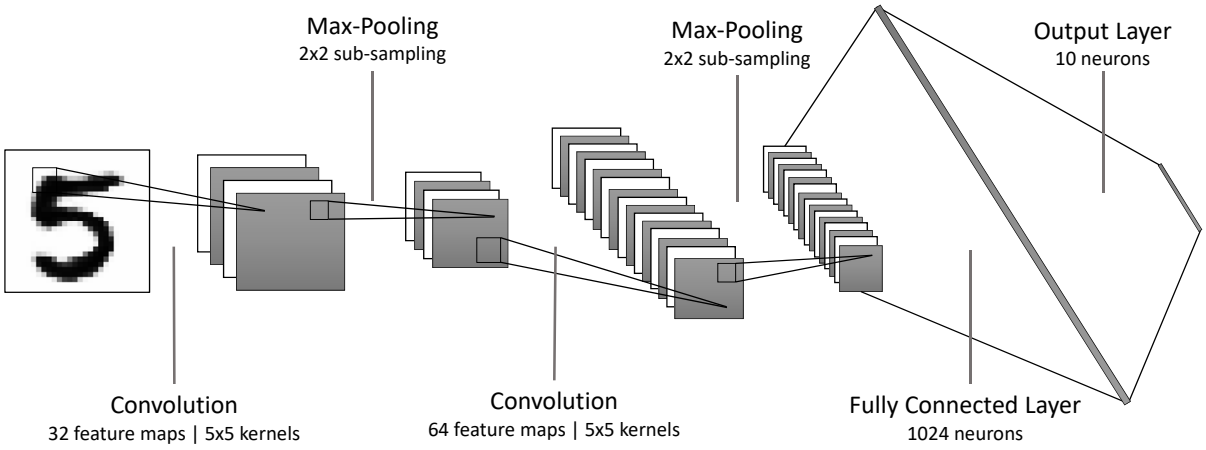


Figure 4.1: Convolutional Neural Network (CNN) used for the TF_CNN and TF_CNN_pruned datasets

As for the NEX, if we equip all systems with the same budget and compare the amount of explorations each one performed, it allows for the evaluation of each one's ability to maximize the search space that is explored when given a predetermined and fixed budget. This is usually correlated with the likelihood of finding a better, as in closer to the optimum, final deployment configuration. We consider one configuration sampled and executed in the cloud as one exploration.

Furthermore, the only stopping criterion considered corresponds to stopping when the exploration budget is over. This allows to focus on evaluating, in a simpler way, the actual key contributions of this work, i.e., the budget-aware, long-sighted planning of the exploration and the early timeout policies, using a fixed horizon for exploring. Hence, it is not pertinent to reason about possible "interferences" due to standard stopping criteria (which are orthogonal to this work) that may decide to stop the exploration phase prematurely, making the dynamics of the various optimizers harder to analyze.

4.2 Datasets

In order to evaluate Lynceus, we used a total of three datasets representative of the training time of Neural Network (NN) models, which were explicitly gathered during this dissertation in order to evaluate the proposed solution.

In Section 4.2.1 we present a detailed description of the datasets gathered in the context of this work, which we call TF datasets. Next, in Section 4.2.2 we introduce a study aimed at shedding light on the characteristics of the datasets used in this work.

4.2.1 Tensorflow Datasets

The TF datasets consider the training of three different Neural Network (NN) models implemented via the TensorFlow [1] framework (a popular ML library) and targeting the MNIST dataset [23] (a standard benchmark for evaluating NNs). This dataset consists of a large database of 28×28 pixels images of handwritten digits. It is composed of $70,000$ images, $60,000$ for training and $10,000$ for testing. All networks utilized are trained through supervised methods, i.e., the networks learn to classify objects based on input-output pairs. Each dataset was obtained through the training of a different NN architecture:

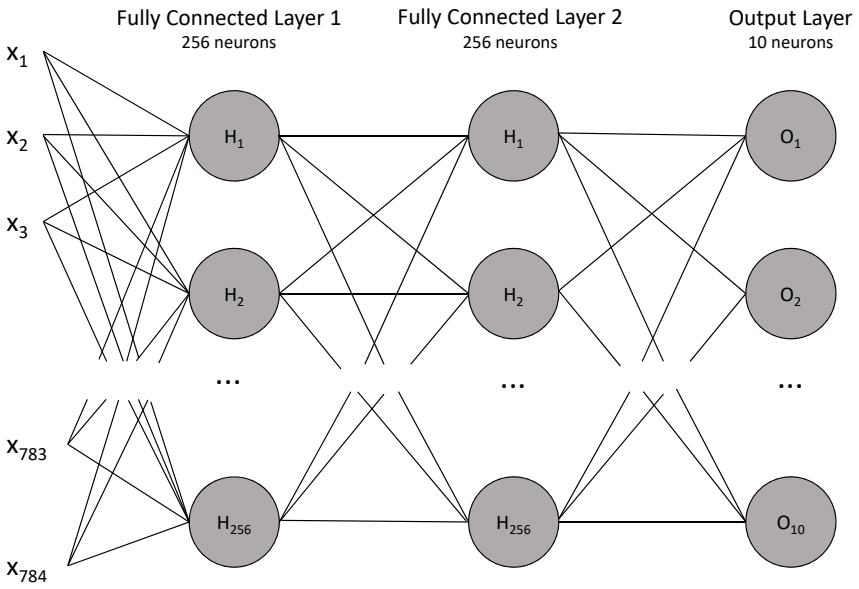


Figure 4.2: Multilayer Perceptron used for the TF_MULTILAYER dataset

- **TF_CNN dataset:** obtained through the training of a Convolutional Neural Network (CNN). This network is a feed-forward network, which starts by automatically extracting features through a series of convolutions and pooling (sub-sampling) operations. Then, these features are given as input to a number of fully connected layers, which ultimately assign a probability for classifying the input. Figure 4.1 depicts the CNN architecture we have used. This architecture was inspired by the general LeNet architecture [44];

- **TF_MULTILAYER dataset:** trains a multilayer perceptron [16]. These networks have at least three layers: an input layer (responsible for reading the input signal), a hidden layer and an output layer (which makes the final prediction). The network is fully-connected, which means that all nodes in a certain layer are connected to all nodes of the following layer with a given weight. These weights are adjusted during the training phase so as to minimize classification error. All nodes in the hidden and output layers are neurons. These neurons use a non-linear activation function, which defines the response of a neuron given an input or a set of inputs. The architecture of the multilayer perceptron employed in the creation of this dataset is represented in Figure 4.2.

- **TF_RNN dataset:** corresponds to the training of a Long Short-Term Memory (LSTM) network [13], a special type of Recurrent Neural Network (RNN) composed of LSTM cells. RNNs include feedback loops in their architecture, enabling them to memorize dependencies about current and previous inputs over an arbitrary time interval. LSTMs introduce the concept of cell states, controlled by additive and multiplicative gates, which enable the network to selectively retain long-term dependencies about a given input, and to adjust the flow of information into and out of a given cell. LSTM cells are also composed by a number of hidden units (or neurons) representing the learning capacity of the neural network. The network utilized to create this dataset is represented in Figure 4.3.

Each dataset is composed of 384 configurations. The configurations considered were composed of combinations of both the parameters of Virtual Machines (VMs) in the Amazon EC2 cloud, as well as parameters affecting the training of NNs (see Table 4.1). Specifically, for the VM parameters we consider the use of four different VM types and eight different values for the total number of cores for the whole

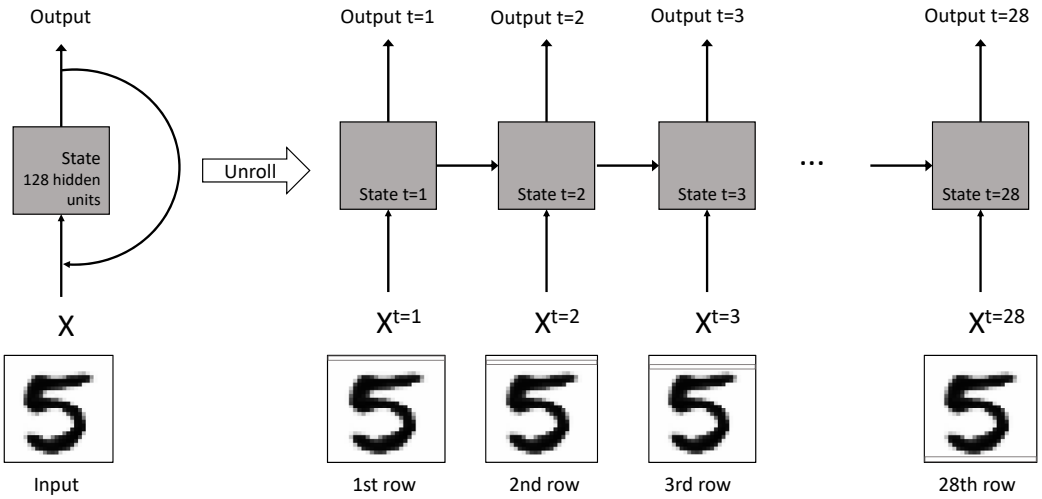


Figure 4.3: Long Short-Term Memory (LSTM) network used for the TF_RNN dataset

Virtual Machines' Parameters		Neural Networks' Parameters		
VM Flavor	**Number of Cores**	**Learning Rate**	**Batch Size**	**Synchronism**
small, medium, xlarge, 2xlarge of the t2 family	8, 16, 32, 48, 64, 80, 96, 112	1e-3, 1e-4, 1e-5	16, 256	synchronous or asynchronous

Table 4.1: Parameters varied to create the space of the configurations

cluster of VMs. Whenever the total number of cores for the cluster exceeds the total number of cores of a single VM of a given type, that means that we consider the deployment of multiple VMs of that type. For what concerns the application dependent parameters, we consider three that affect the behavior of the algorithm (Stochastic Gradient Descent [58, 39, 9]) used to train the models, namely: three values of the learning rate, two values for the batch size and whether the training takes place synchronously or not [16]. These parameters hold for all datasets except for the TF_CNN dataset which considers only six different values for the number of cores($\{32, 48, 64, 80, 96, 112\}$), and therefore is composed of 288 configurations.

To build our datasets we gathered data by simulating the behaviour of the Executor upon being ordered to run a certain job in a given configuration. We resorted to Python and to boto3 (the Amazon Web Services (AWS) SDK for Python) for implementing the Executor's logic and to run the selected configurations on AWS. The Executor is emulated by a machine which acts as job coordinator and which is able to deploy a given job in a selected configuration. For each exploration there is a total of $N + 2$ VMs that are deployed on AWS. The N VMs correspond to the number of machines specified by the configuration that is being tested. The remaining 2 VMs are helpers. One embodies the parameter server for the NN model. In the parameter-server framework, a parameter server keeps records of the NN parameters that have been continually improved by the workers. The other acts as bookkeeper, measuring the model's accuracy at predefined time intervals (each 30 seconds, for all datasets). The necessity for the bookkeeper VM stemmed from the fact that querying the model to obtain an accuracy measurement translates into a non-negligible amount of execution time which would impact the speed of model training should these measurements be collected from a dedicated worker machine.

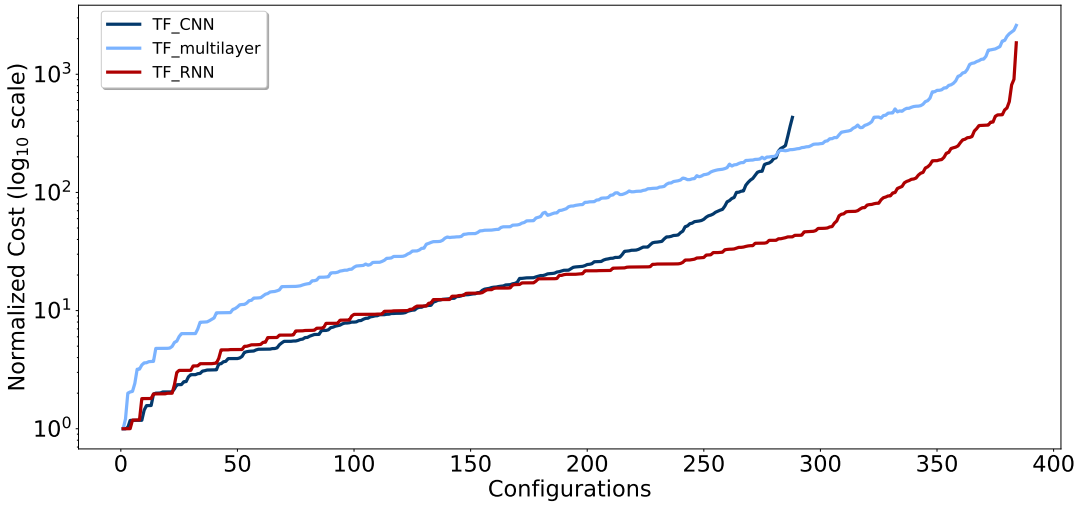


Figure 4.4: Datasets' complexity

4.2.2 Analyzing the Datasets

In this section we aim at shedding light on the complexity of the considered datasets. A first aspect to consider, to this end, is the cardinality and dimensionality of the search spaces considered by the various data sets, which vary from 288 configurations (CNN) to 384 (RNN and MULTILAYER) configurations distributed over 6 dimensions.

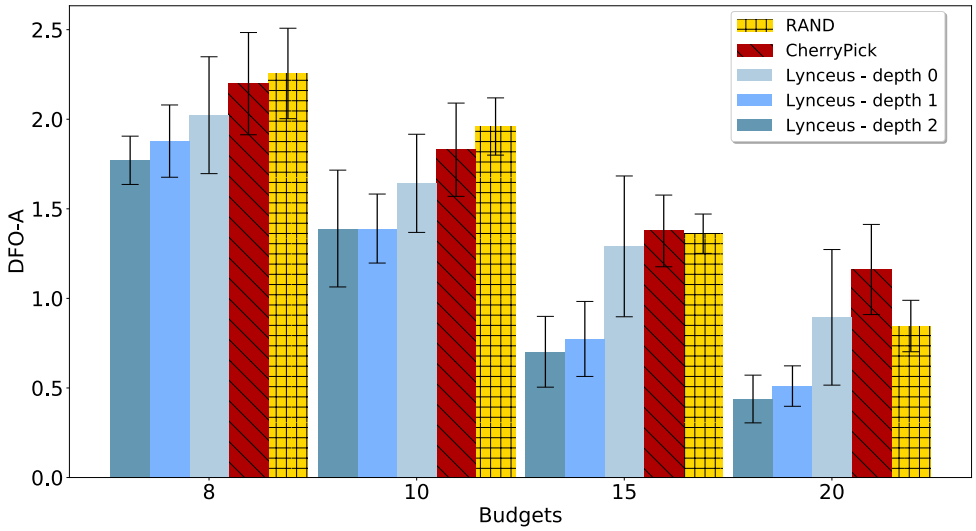
In order to quantitatively assess the complexity of each dataset, in Figure 4.4 we report a plot that illustrates the distribution of the Distance From Optimum (DFO) for all the configurations (ordered from best to worst) for each dataset. The DFO is defined as the cost of a configuration normalized to the cost of the cheapest one for that dataset.

The plot shows that the TF datasets have very few near-optimal configurations. In fact, for the TF_-MULTILAYER dataset, the second best configuration is already 20% away from optimum (note that the y-axis is expressed in log scale). A steeper slope translates higher difficulty and the TF_MULTILAYER dataset shows quite a steep slope close to the optimum solution. Therefore it is expected that the tested systems, when evaluated with these datasets, are more likely to output final configurations further away from the optimum.

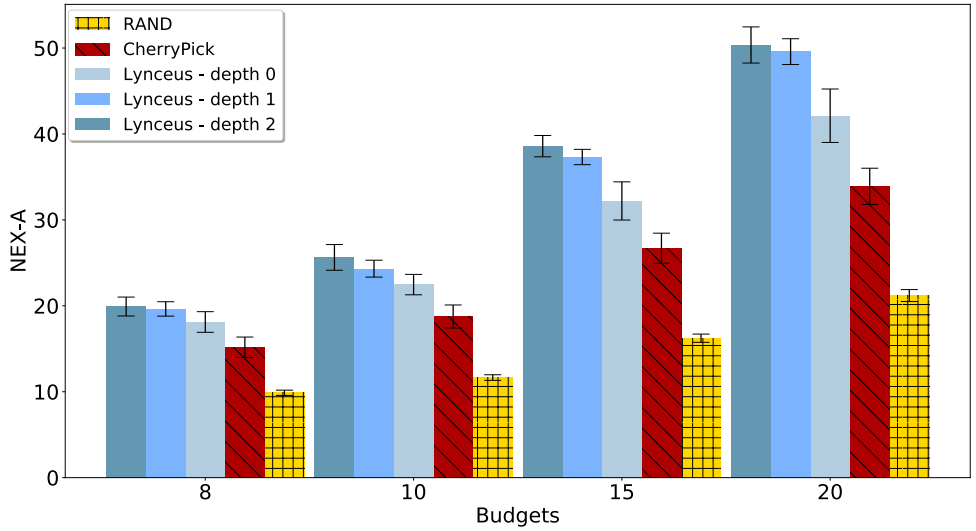
4.3 System Implementation and Experimental Setup

The following paragraphs describe the choices taken with respect to the system implementation and to the chosen settings for running the experiments.

System Implementation. Lynceus was developed using Java, namely leveraging existing libraries for the implementation of the performance model that predicts the next configurations to explore. Our system uses the random tree algorithm available in the Weka software package [33]. This algorithm builds a decision tree considering K randomly chosen attributes at each node and performs no pruning. We build 10 of these trees to generate an ensemble, similarly to a typical random forest implementation, since the attributes of each tree are random. The ensemble serves as a performance model to predict



(a) Average of the DFO



(b) Average of the NEX

Figure 4.5: Average of the NEX and DFO using the dataset TF_CNN

the costs of the configurations. For the implementation of the in-depth simulation, we ensure the long-sightedness of our approach by implementing the *Gauss-Hermite* quadrature.

Experimental Setup. For the experiments without the timeout policy, the initial models of all systems are bootstrapped with 5 initial configurations, randomly selected, which corresponds to roughly 2% of the search space. For the experiments with the timeout policy, the models are bootstrapped with 1 initial sample. The scarcity of initial points augments the difficulty of finding the optimal configuration, since the existent knowledge to guide the search is fairly reduced. The parameters β and γ of Lynceus were attributed values 0.99 and 0.9, respectively, as proposed by Remi et.al [43]. Regarding the look-ahead factor, the values $d = \{0, 1, 2\}$ were considered. A look-ahead of 0 cancels out Lynceus' ability of predicting/simulating the future, allowing for the establishment of a baseline for comparison with CherryPick. Since Lynceus is budget-aware, and since we wanted to test its ability to find the optimal configuration for deploying a user specific application in different scenarios, another of the parameters that was varied across all experiments was the budget. We considered budgets of $\{2, 4, 6, 8, 10, 15, 20\}$ times the average budget of a configuration for the experiments, although budgets $\{2, 4, 6\}$ were only considered for the experiments with 1 initial sample. We considered one situation of user-defined QoS

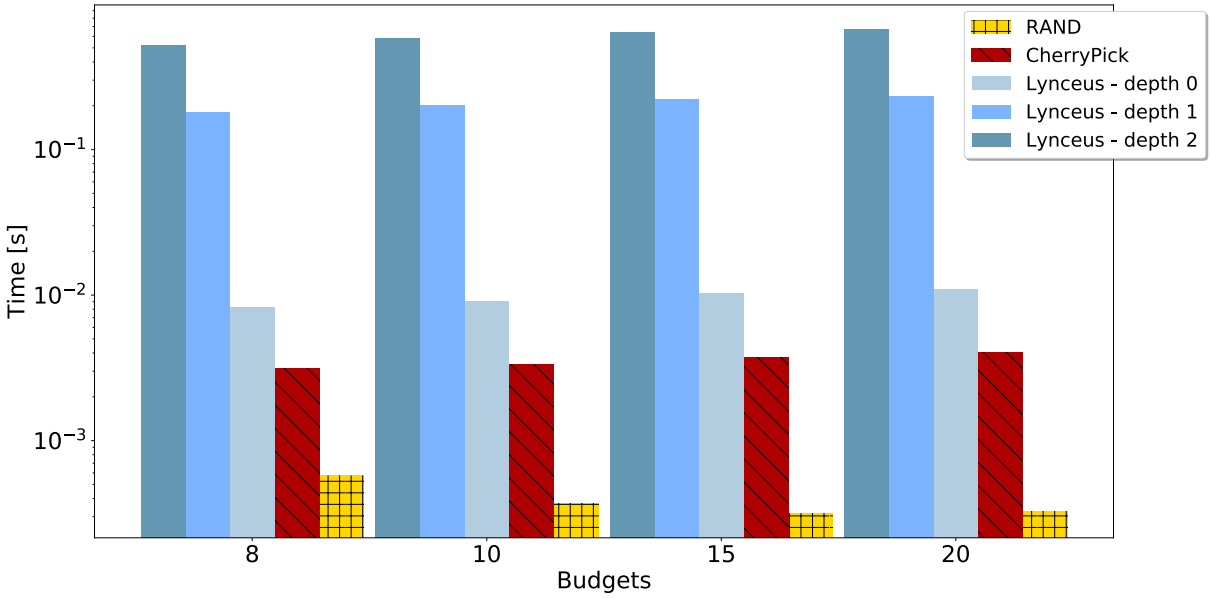


Figure 4.6: Average time corresponding to 1 exploration with the TF_CNN dataset

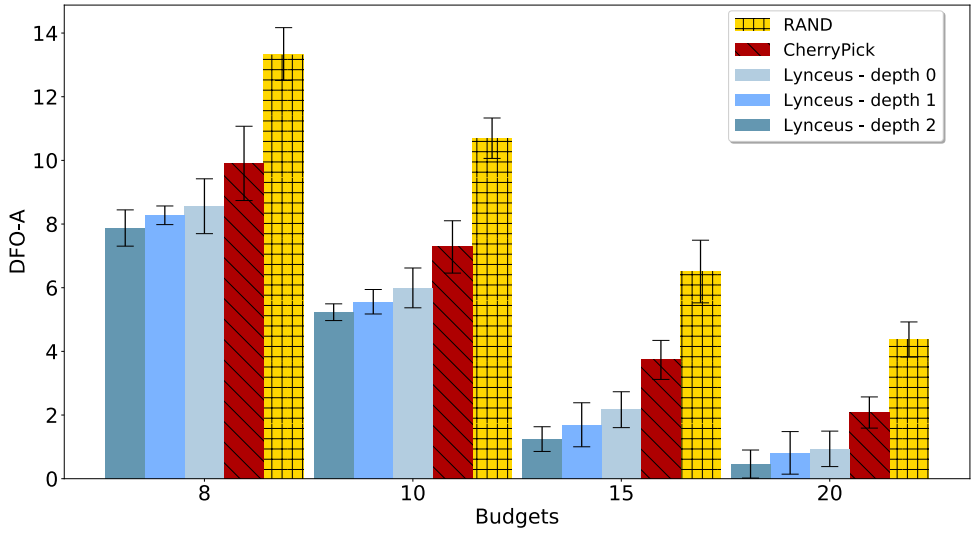
constraints that corresponds to *T_MAX* $= running\_time(fastest\_config) \times (1 + max\_time\_perc)$ minutes. The max_time_perc variable was set to 10. For lack of time, we didn't perform a sensitivity analysis on this variable. Lastly, we perform 500 executions of all experiments with the TF datasets without the timeout policy. For all other experiments 50 runs are executed. In both cases, the results of the runs are averaged to compare the three approaches. The standard deviation shown in the TF plots in Section 4.4 is computed resorting to 5 batches of 100 points each. The values of the DFOs and of the NEXs of each batch are averaged, resulting in 5 averages for each parameter, one for each batch. These averages are the ones with which the standard deviation is then computed. All experiments were executed in machines running Linux Ubuntu 16.04 LTS, which were equipped with an Intel Xeon E5-2648L CPU and with 32GB RAM.

4.4 Quality of the Final Configuration

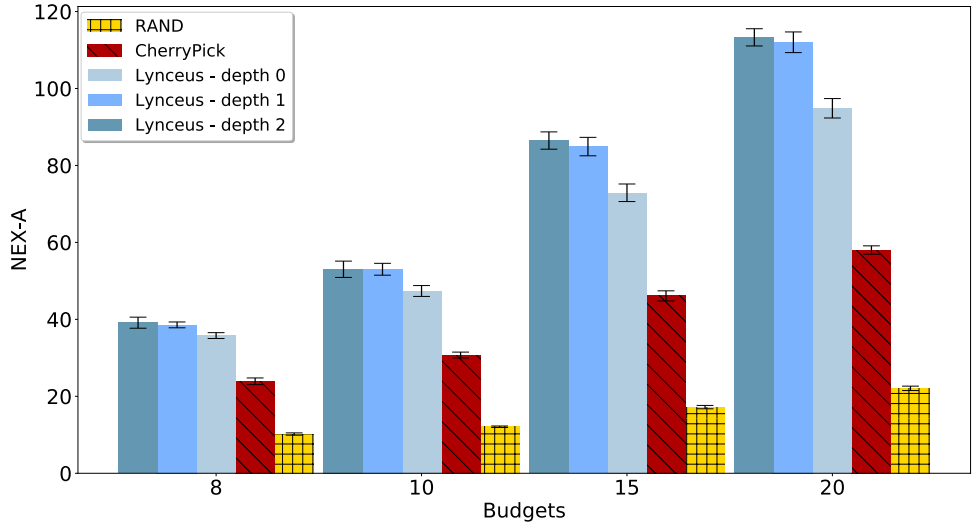
All approaches output a final prediction regarding the selected configuration to run a user-specific application in the cloud setting. The quality of the final recommended configuration can be measured by its distance to the optimal one. Furthermore, the number of explored configurations is expected to be correlated with better final predictions. To validate this assumption and to evaluate Lynceus' predictions, this section presents the results obtained for all datasets regarding both evaluation metrics and considering the two systems and the Random approach. All results presented in this section do not feature the timeout policy optimizations, which are evaluated in Section 4.5.

4.4.1 TF_CNN Dataset

Figure 4.5 displays the average of the 500 executions performed with the TF_CNN dataset. Observing the NEX (Figure 4.5b), it is visible that, as expected, higher budgets allow for more explorations. For instance, Lynceus with look-ahead 2 and budget 8 performs 20 explorations and doubles this value, approximately, with budget 15, performing 39 explorations. Furthermore, we can conclude that having a



(a) Average of the DFO



(b) Average of the NEX

Figure 4.7: Average of the NEX and DFO for the three approaches using the dataset TF_MULTILAYER

predictive model to guide the search is rather helpful in getting an increased knowledge of the search space, since both CherryPick and Lynceus explore more than the Random approach with any budget. For budgets 8 and 10, Lynceus with look-ahead 0 and CherryPick are very similar in terms of NEX however, for budgets 15 and 20, Lynceus is able to give a better use to the budget and therefore explore more.

Nevertheless, one would expect that more knowledge of the search space, provided by a higher number of performed explorations, would be translated into better, closer to the optimum solutions. In fact, a good model fed by a larger number of explorations is more likely to find the optimal or near-optimal configuration. However, and as we can observe in Figure 4.5a, that is not always the case. For instance, for budget 10, Lynceus explores approximately the same number of configurations for all look-aheads. Despite that, the final results for look-ahead 2 are equally as good as those with look-ahead 1. This fact may be due to an inaccurate initial model because of the random initial samples. Nonetheless, Lynceus is able to find configurations closer to the optimum than the other approaches, for all budgets. As the budget increases, the quality of the final configurations selected by all approaches improves.

One other relevant conclusion that can be drawn from Figure 4.5a concerns the difference between the DFOs exhibited by CherryPick and Lynceus with look-ahead 0. This is due to the function that estimates the quality and the expected contribution to the performance model of exploring a given configuration. While Lynceus considers the ratio between the expected contribution to the performance model and the cost of the configuration, CherryPick only considers the expected contribution of an exploration. For example, consider two configurations with the same expected contribution to the model and different costs: $C1$ with $cost(C1) = 10$, $C2$ with $cost(C2) = 5$ and $EIc(C1) = EIc(C2)$. For CherryPick, these two configurations would be equally good candidates for the next exploration, since they offer the same EIc. However, Lynceus would choose configuration $C2$ as it offers a better quality/cost ratio.

For measuring the computational complexity of the tested approaches, we computed, for each execution, how long they took to make the prediction of the next configuration to explore and then averaged those times. Figure 4.6 displays the average execution time of one exploration for each approach and for each budget. As expected, Lynceus, which adopts a more complex optimization procedure, incurs larger computational costs than the other counterparts. Yet, the average latency to predict which configuration to explore next is on the order of 0.5 seconds, thus making Lynceus a viable solution in practice, considering that many typical cloud jobs last several hours or days [67].

4.4.2 TF_MULTILAYER Dataset

Figure 4.7 shows the average of all executions of the TF_MULTILAYER dataset. With this dataset, just like with the TF_CNN dataset, more knowledge or more explorations is also not always directly correlated with better final solutions. Additionally, and as can be observed in Figure 4.7a, there are also discrepancies in the quality of the solutions of CherryPick and of Lynceus with look-ahead 0. This is due to the different functions employed by each system to select the next configuration to explore. Furthermore, we also notice that, for all budgets, although there are always more explorations for Lynceus with look-ahead 1, Lynceus with look-ahead 0 finds configurations closer to the optimum. All budgets show that a higher look-ahead helps finding better final configurations. Random's results are consistent with the NEX results in the sense that less explorations are translated into a higher distance from the optimum. CherryPick, that explores less than Lynceus, always finds configurations further away from the optimum.

In Figure 4.7b the Number of Explorations (NEX) supports the previous conclusion that having a predictive model to guide the search boosts the knowledge of the search space, since both CherryPick and Lynceus explore at least two times more than Random. As expected and as observed in the results of the previous dataset, for higher budgets, all approaches are able to attain more explorations. In comparison with CherryPick, Lynceus achieves always at least more 50% explorations. There seems to be a trend with this dataset, since Lynceus with longer look-aheads, always performs the same or more explorations than Lynceus with a smaller or with no look-ahead.

The computational complexity of each approach for making a prediction of the next configuration to explore when running the TF_MULTILAYER dataset is displayed in Figure 4.8. Although Lynceus remains the slowest, as with the previous dataset, the overhead it introduces is negligible. This is especially true in a situation where the time it takes to make a prediction is not paramount for the user, particularly when the improvements achieved by Lynceus are considered. For instance for budget 15 and look-ahead 2, the solution found by Lynceus is more than two times closer to the optimum than the one found by CherryPick and three times better than the one found by the Random approach.

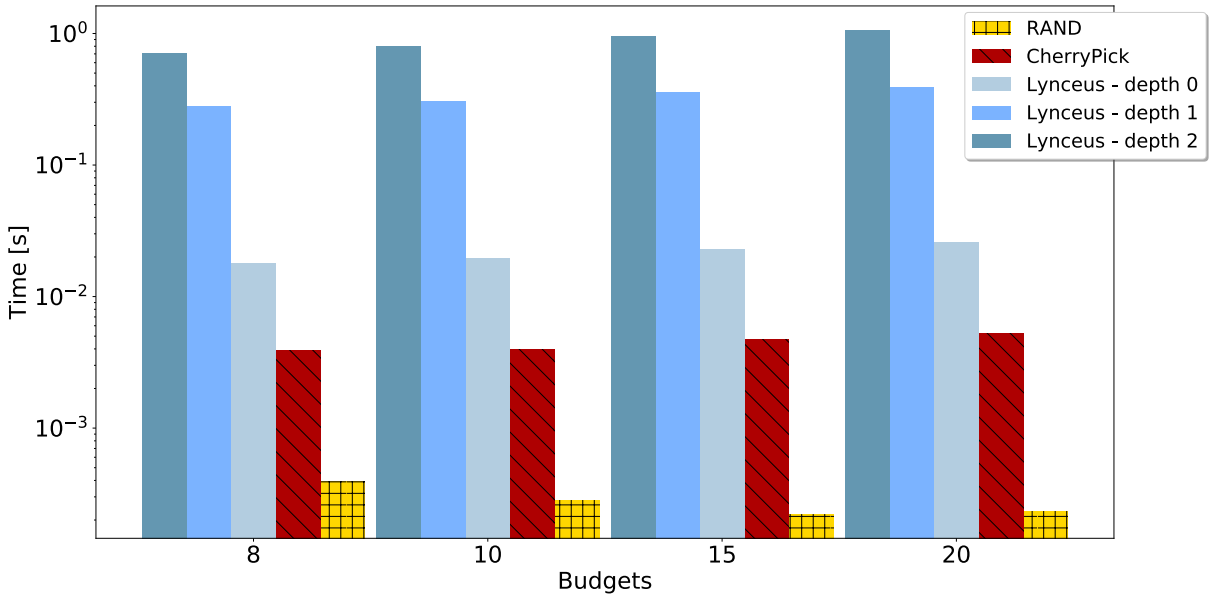


Figure 4.8: Average time corresponding to 1 exploration the TF_MULTILAYER dataset

4.4.3 TF_RNN Dataset

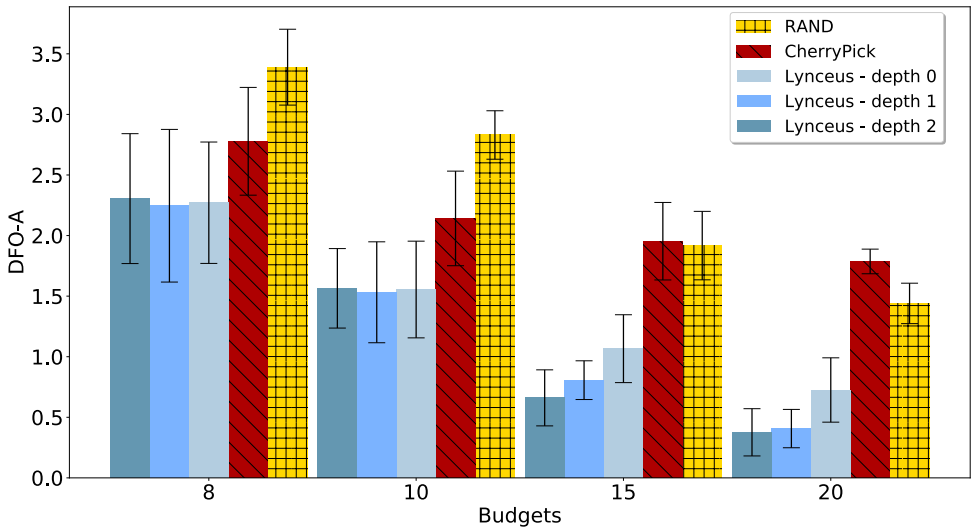
Figure 4.9 shows the average of the executions of the algorithm with the TF_RNN dataset. The DFO results (Figure 4.9a) further confirm that Lynceus finds better configurations than CherryPick and than the Random approach with the TF datasets. For budget 20 and look-ahead 2, Lynceus finds solutions 38% away from the optimum, on average.

The NEX results (Figure 4.9b) are consistent with the results from the previous two datasets. The Random approach has the lowest NEX, followed by CherryPick and then Lynceus, having the most explorations. With this dataset as well, Lynceus and CherryPick always perform at least the double of the Random approach's explorations. With smaller budgets, Lynceus' model is fed with few points, thus rendering it rather inaccurate which harms the simulations when Lynceus is used with look-aheads. With look-ahead, in fact, Lynceus plans the next exploration step based on simulating future explorations using the model's output. It is unsurprising that the more accurate the model is (as it is the case with larger budgets that allow to explore a larger number of configurations), the larger the benefits stemming from the use of non-myopic policies.

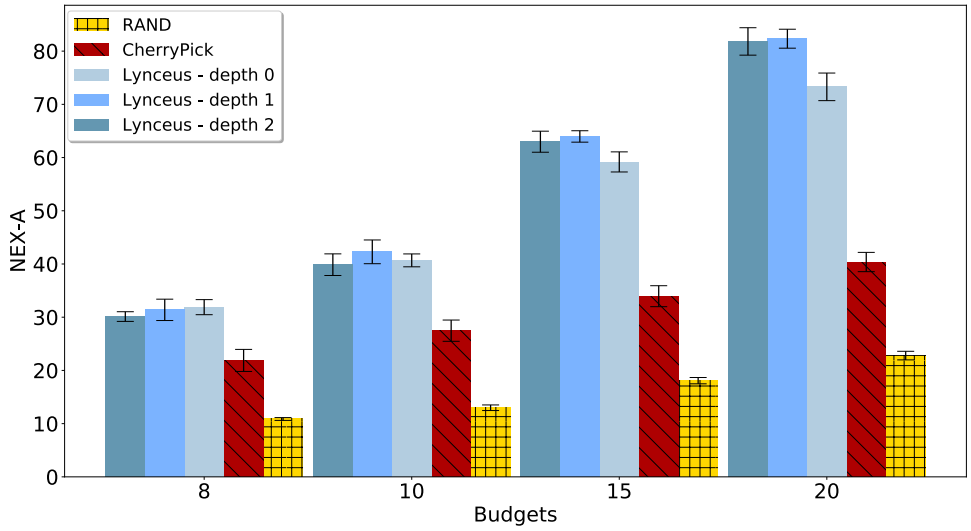
Computational complexity results for this dataset (Figure 4.10) are in line with the results of the previous one. Not only are the differences between CherryPick and Lynceus with look-ahead 0 much less meaningful than with the TF_CNN dataset, but also for budget 15 and look-ahead 2 the solution found by Lynceus is more than 2 times better than the one found by CherryPick. The Random approach, although the fastest, remains worse than Lynceus and, in most cases, worse than CherryPick as well.

4.4.4 Discussion

Comparing and observing the DFO plots of all datasets we notice a tendency for the best configurations to be found resorting to Lynceus with look-ahead 2. This shows that having a long-sighted, budget-aware strategy, instead of a greedy strategy, to select the next configuration to explore allows to find better configurations. Furthermore, the discrepancies in the quality of the solutions of CherryPick



(a) Average of the DFO



(b) Average of the NEX

Figure 4.9: Average of the NEX and DFO for the three approaches using the dataset TF_RNN

and of Lynceus with look-ahead 0 emphasize the importance of the function that selects the configurations to be explored. The use of EIc as acquisition function in Cherrypick leads, on average, to identifying configurations 50% further away from the optimum than using Lynceus with look-ahead 0, which uses, as acquisition function, the ratio of the EIc over the expected exploration cost.

All results show that Lynceus, with the same budget as the other approaches, although consistently slower, is able to explore more of the search space and find solutions that are closer to the optimum. Furthermore, by analyzing all computational complexity plots and, simultaneously, comparing DFOs, the plots suggest that the quality of the final configurations selected by Lynceus makes up for the time overhead it induces. Running Lynceus on an common off-the-shelf hardware allows for better results than with any of the other approaches, although not instantaneously. Nonetheless, an overhead in the order of a few seconds is perfectly affordable given that many cloud jobs last hours/days/weeks.

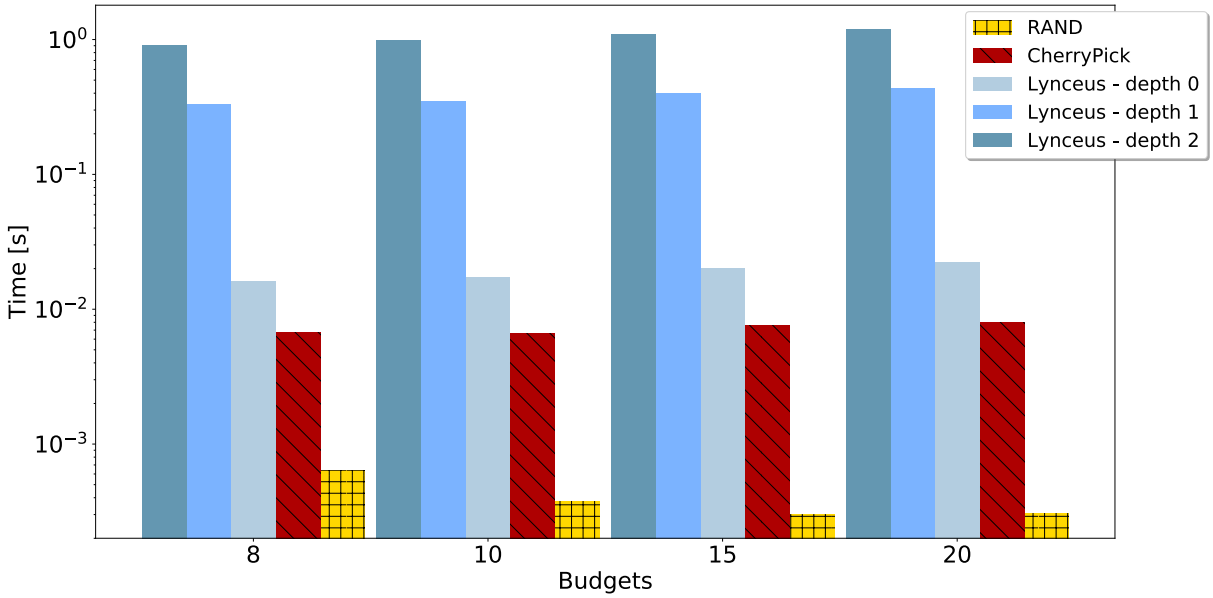


Figure 4.10: Average time corresponding to 1 execution the TF_RNN dataset

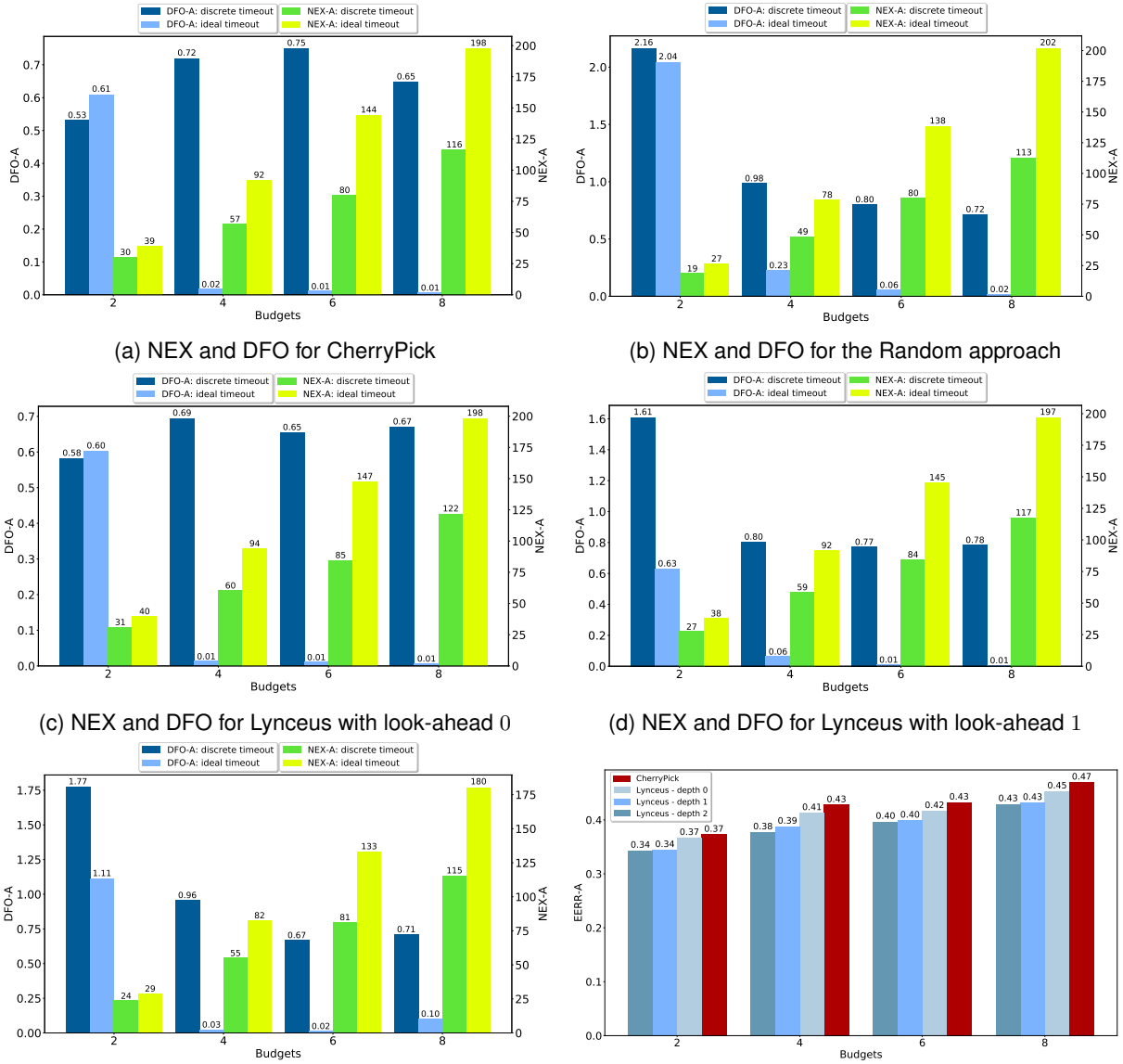
4.5 Improvements Attained due to the Timeout Policy

This section details the results and improvements obtained through the application of the timeout policy introduced in Section 3.4. This policy considers time intervals of 30 seconds to measure the accuracy of the jobs. These experiments were run with 1 initial sample for bootstrapping the model and with budgets $\{2, 4, 6, 8\}$. In fact, as a consequence of the introduction of this optimization, the exploration process becomes much more cost-effective. Therefore, when using larger budget values (e.g., the ones used in the previous section), in most scenarios the considered optimizers would be able to explore the whole search space, and, trivially, always identify the optimal configuration. Due to time constraints we were unable to evaluate the TF_MULTILAYER dataset with this policy. Also due to time constraints, we were unable to evaluate the TF datasets without the timeout policy for these budget values and initial number of sampled points. Therefore, we do not show the DFO and NEX results without timeout in these plots. However, considering the previous results obtained with budget 8 and 5 initial samples (Section 4.4), it is reasonable to assume that using budget values lower or equal to 8 and a single initial sample, all approaches would find configurations more than 150% away from the optimum.

The ideal timeout policy assumes that the progress of the application is known at each instant, which does not correspond to a real world setting, since each accuracy measurement, for instance in the case of a NN training job, takes a non-negligible time interval. With this policy, the linear model that estimates the costs upon completion is considered to be perfect and, therefore, the performance model is updated with the exact cost. Thus, no estimation errors are introduced in the performance model. This policy is used as a baseline to assess the viability of using a simple linear based estimation of the cost of executing the job until completion.

4.5.1 TF_CNN Dataset

Figure 4.11 depicts the improvements achieved due to both the ideal and the discrete timeout policies with the TF_CNN dataset. Results shows that all approaches benefit from both timeouts, which allows them to increase significantly the total number of explored configurations and yield higher quality



(a) NEX and DFO for CherryPick

(b) NEX and DFO for the Random approach

(c) NEX and DFO for Lynceus with look-ahead 0

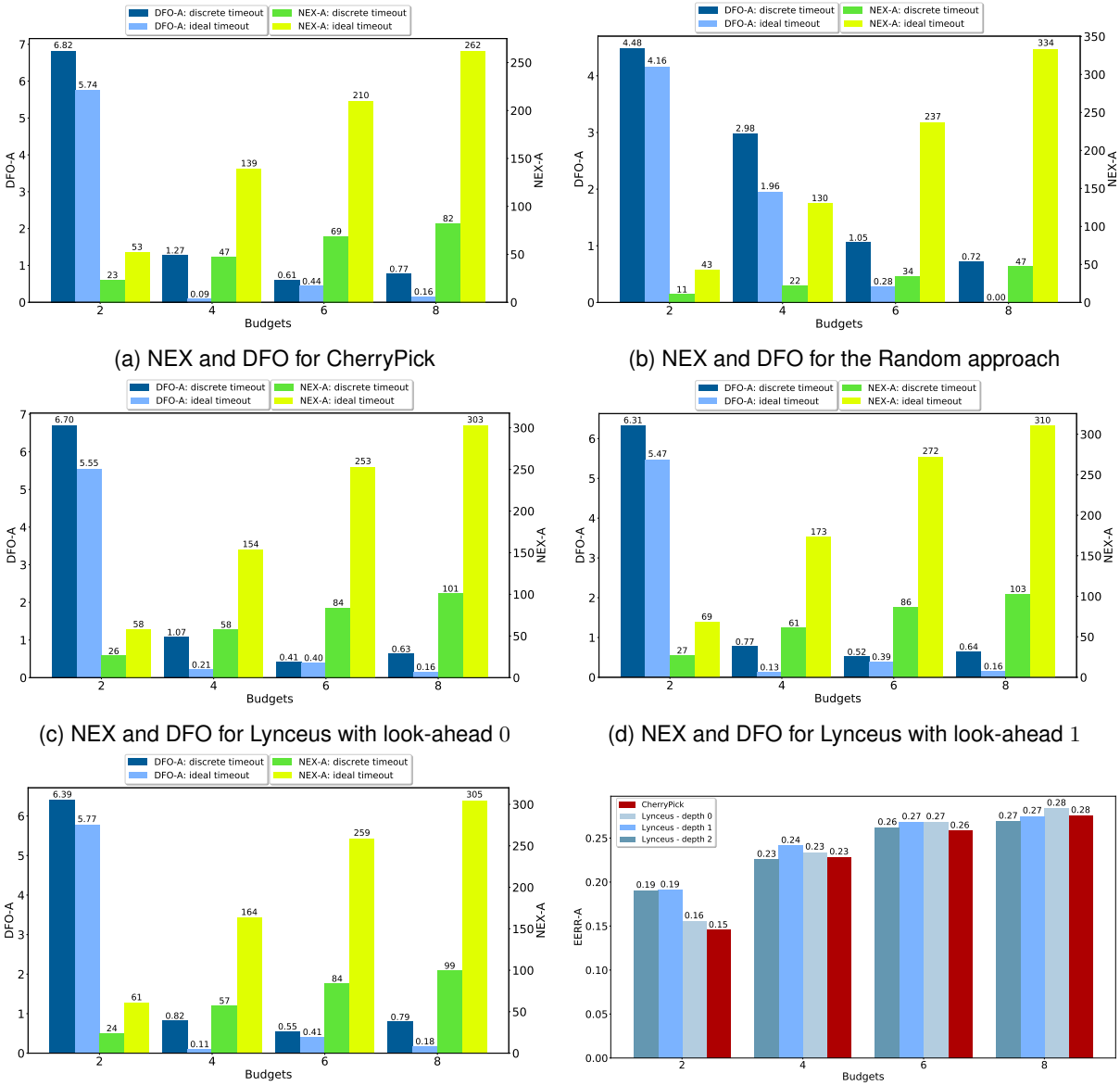
(d) NEX and DFO for Lynceus with look-ahead 1

(e) NEX and DFO for Lynceus with look-ahead 2

(f) Average estimation error of the linear model

Figure 4.11: Comparison of the NEX and DFO without timeout and with both the ideal and the discrete timeouts for all approaches and for the TF_CNN dataset

configurations at the end of the search process, when compared to the case in which no timeout is used. With respect to the NEX, we observe that it increases with the budget, for all approaches and for both timeouts. Moreover, with the ideal timeout and with budget 8, both CherryPick and Lynceus with look-ahead 2 explore approximately $2/3$ of the search space, whereas with the discrete timeout they explore around 40% of the space of configurations. In terms of the DFO, although the ideal timeout is extremely close to the optimum for higher budgets (only 1% and 2% away from the optimum), it also fails in some scenarios and especially for smaller budgets. With the discrete timeout, the results also display higher benefits for CherryPick than for Lynceus with either look-ahead 1 or 2. For instance for budget 4, the configuration found by CherryPick is 72% away from the optimum while Lynceus with look-ahead 2 is 96% away. We argue that this may be due to the fact that the liner model that estimates the cost of running the job until completion is rather inaccurate which leads to the performance models of both CherryPick and Lynceus to be polluted with erroneous data. The average of the mean absolute percentage error of this model is depicted in Figure 4.11f. The error remains approximately constant, around 40%, for



(a) NEX and DFO for CherryPick

(b) NEX and DFO for the Random approach

(c) NEX and DFO for Lynceus with look-ahead 0

(d) NEX and DFO for Lynceus with look-ahead 1

(e) NEX and DFO for Lynceus with look-ahead 2

(f) Average estimation error of the linear model

Figure 4.12: Comparison of the NEX and DFO without timeout and with both the ideal and the discrete timeouts for all approaches and for the TF_RNN dataset

CherryPick and for all look-aheads of Lynceus equipped with the same budget. We argue that, when using non-zero look-ahead values, Lynceus is inherently more sensitive to the fallacies of the underlying cost model, given that it relies on the model to simulate the effects of multiple future explorations. This effect is particularly exacerbated when the available budget is quite limited: in this case, in fact, the cost model used by Lynceus can rely only on a quite limited number of training data, and is, as such, more prone to be polluted by errors introduced by the employment of the discrete early timeout policy. Indeed, at larger budget values (6 and 8), the accuracy gap between Lynceus, with look-aheads 1 or 2, and CherryPick reduces significantly, as the two solutions, when equipped with the early timeout policy, tend to identify configurations with similar quality. The Random approach is not affected by the estimation errors of the linear model since it does not have a performance model to select the next configuration to explore.

4.5.2 TF_RNN Dataset

Figure 4.12 shows the results for the ideal and discrete early timeout policies, when considering the TF_RNN dataset. The trends exhibited by this dataset are similar to the ones observed with the TF_CNN dataset. As expected, the ideal timeout policy leads to larger gains with respect to the discrete timeout policy also with dataset. This has two main implication. First, it suggests that that the errors introduced by a simple linear model to estimate the cost of early timed out configurations do introduce a perceivable degradation of the DFO. For instance, Lynceus with look-ahead 0 and budget 4 finds solutions 21% away from optimum and explores 154 configurations with the ideal timeout, while with the discrete timeout the final configurations are, on average, 107% away from optimum and 58 explorations are performed. This consideration motivates future work aimed at investigating the use of alternative/more sophisticated estimators. Second, these results confirm that, even when coupled with simple linear estimators, the proposed early timeout policy still provides remarkable gains when compared with the no-timeout scenario. For instance for budget 8, CherryPick with no-timeout finds configurations 254% away from the optimum while with the early timeout policy this is reduced to 16%. An interesting conclusion that can be drawn by analyzing the results obtained with this dataset is that, since the average error of the linear model (Figure 4.12f) is slightly lower for this dataset than for the previous one (around 20%), Lynceus with the discrete timeout and with look-aheads 1 and 2 is better than CherryPick. Results show that, for instance for budget 4, CherryPick finds solutions 127% away from optimum with 47 explorations while Lynceus with look-aheads 1 and 2 is 77% and 82% away from optimum, with 61 and 57 configurations explored, respectively.

Finally, the results obtained when coupling the early timeout policy with a simple random optimizer (see Figure 4.12b) confirm that the proposed time out policy is particularly effective even when coupled with model-less optimizers. In this case, given the model-free nature of the optimization process, though, the difference in the performance of the ideal versus discrete timeout policies is not imputable to potential inaccuracies in the estimation of the exploration cost of timed out configurations. Conversely, the relatively higher performance of the ideal early timeout policy is due to the fact that it assumes the possibility of evaluating continuously whether the exploration of a configuration should be prematurely interrupted (either because it is found to have a higher cost than the current optimum or because it has violated the user defined QoS constraints). As discussed in Section 3.4, though, in real settings, the predicate for deciding whether to time out an exploration can only be evaluated periodically (each 30 seconds in this experiment).

Summary

This chapter detailed the characteristics of the datasets built for evaluating Lynceus and reported the results of an experimental study aimed at assessing Lynceus's ability to optimize complex, realistic cloud applications by comparing it with two baseline solutions: a state-of-the-art technique, CherryPick [2], based on a myopic/not budget-aware SMBO approach, as well as a simple random optimizer. Overall, the experimental results have shown that the long-sighted, budget aware SMBO methodology used by Lynceus enables significant gains in terms of reduced DFO and increased number of explored configurations, at parity of budget, when compared with CherryPick. In fact, Lynceus finds configurations almost 5 times closer to the optimum (Figure 4.9a) and explores more than two times more than CherryPick (Figure 4.9b), in the best case.

In order to evaluate the effectiveness of the proposed early time out policies, we tested whether

its usage could be beneficial not only when coupled with Lynceus' optimization policy, but also when used in combination with the two considered baselines. To this end, we considered also an idealized implementation of the timeout policy, which assumes that the progress of the application is known at each instant and that the model for estimating the cost of executing the job until completion is able to make accurate predictions, not incurring errors, and therefore not polluting the performance model with inaccurate predictions. The discrete timeout policy was shown to provide solid gains except for the lowest budgets. For instance CherryPick with budget 8 and for the TF_CNN dataset has the DFO reduced from 187% to 65% due to the discrete timeout. With the lowest budgets, the errors introduced by estimating the actual cost of fully running an application in a timed out configuration polluted the performance model, leading to inaccurate predictions and therefore to worse final configurations. For the TF_RNN dataset, Lynceus with look-ahead 2 and budget 2 finds configurations 639% away from the optimum. As for the ideal timeout policy, the Random approach's results (Figures 4.11b and 4.12b) allow us to quantify the speed-ups brought by timing out the explorations as soon as they become unfeasible. Besides quantifying these speed-ups, CherryPick's and Lynceus' plots quantify as well the benefits attained due to updating the performance model with the exact cost of running the job until completion. Hence, for CherryPick's and Lynceus' plots, the gaps between the DFOs are bigger than for the Random approach's plots. In fact, for the dataset TF_RNN and with budget 2, the gap between the ideal and discrete timeouts for Random is approximately 30% while for CherryPick is more than 100%.

When compared with the ideal timeout policy, the discrete timeout policy clearly achieves lower gains, motivating the need for future research aimed at striving to fill this gap, e.g., by using more accurate predictive models and/or adapting dynamically the monitoring frequency, and increasing/decreasing it when there are larger/smaller risks for a configuration to be subject to an early timeout.

The next chapter concludes this document by summarizing the main findings of this thesis and introducing some directions for future work.

5 Conclusions and Future Work

In this dissertation, we introduced Lynceus, a system for optimizing the selection of configurations for the execution of cloud applications, under the pragmatical assumption of having available a fixed budget for the whole self-tuning process. Lynceus incorporates explicitly, and for the first time in the context of self-tuning of cloud applications, the notions of finite budget for exploration and of planning of future explorations. To achieve this, Lynceus leverages innovative techniques in the field of Bayesian Optimization (BO) with finite budget [42] and with look-ahead [43]. The main challenge that had to be overcome was deciding how the exploration of the configurations should proceed so as to ensure that each exploration improved the model the most.

Through an extensive experimental evaluation based on realistic datasets, we demonstrated how the proposed solution is able to consistently identify better solutions than Cherrypick, a recent state-of-the-art optimizer for cloud applications, across heterogeneous datasets encompassing a popular cloud based application, i.e., training of machine learning jobs. We also show how our long-sighted approach enables improvements over greedy search strategies, namely Lynceus with look-ahead 2 and budget 20, in the best case, finds configurations almost 5 times closer to the optimum than CherryPick equipped with the same budget.

The timeout policy results show that by reducing an experiment's duration it is possible to find better final configurations with lower budgets. Moreover, this policy introduced significant gains, allowing the DFO to decrease to approximately 50% from the optimum for Lynceus with look-aheads 1 and 2, for budget 6 and with the TF_RNN dataset. We also showed how the linear model utilized with the discrete policy is far from optimum, having errors of 50%.

As for future research directions, we believe that a multitude of possible optimizations to further upgrade Lynceus can be implemented. These optimizations are concerned with the initial bootstrapping of the model, with the use of local search heuristics to improve Lynceus' computational complexity and with the development of more complex models for the early timeout.

Our algorithm uses a model-driven approach to select which configurations to explore, but, before the model can be instantiated, one needs to gather an initial set of samples to boostrap the knowledge of the model. In our current implementation, for simplicity, we use a randomized approach that selects configurations uniformly at random within the configuration space. Clearly, this approach can be subject to various shortcomings. In particular, it may lead to sampling overly expensive configurations, reducing significantly the budget available for the model-based exploration phase (which is arguably more effective). Further, since it is desirable to use a very limited number of initial samples, a purely randomized approach can lead to miss sampling relevant regions of the configuration space — hindering the quality of the resulting model. The latter issue can be obviated by recurring techniques such as Latin Hypercube Sampling (LHS) [62], which strives to uniformize the density with which the various sub-regions of a multi-dimensional space are sampled.

The use of local search heuristics, such as Hill-Climbing, Simulated Annealing or Genetic Algorithms may be of assistance to further reduce Lynceus' computational complexity. These heuristics would

be applied whenever in Algorithms 2 and 4 we need to compute the EIc or the *Utility/Cost* for every unexplored configuration. By employing one of the above search heuristics, the computation of the EIc would be restricted to smaller subsets of the set of unexplored configurations, with direct gains in terms of computational efficiency.

The comparison with the ideal timeout policy showed that there are still margins for improving the early timeout policy. One possible future direction could be the development of more complex models to estimate the cost of running a job until completion. Another possibility is to develop additional, and not so ideal, policies to assess whether the gap between the ideal and the discrete policies is due to the non-immediate timeout or to the errors introduced when predicting the expected cost of fully running the job in that configuration. A possible additional policy could be timing out immediately but not being able to update the performance model with the exact cost of running the job until completion.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "Tensorflow: a system for large-scale machine learning". In: *Proceedings of the 12th Symposium on Operating Systems Design and Implementation*. Vol. 16. Savannah, GA, USA, 2016, pp. 265–283.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. Boston, MA, USA, 2017, pp. 469–482.

[3] Amazon. *Elastic Compute Cloud*. `https://aws.amazon.com/ec2/`.

[4] Peter Auer. "Using confidence bounds for exploitation-exploration trade-offs". In: *Journal of Machine Learning Research*. Vol. 3. Nov. JMLR, inc. 2002, pp. 397–422.

[5] Richard Bellman. "The theory of dynamic programming". In: *Bulletin of the American Mathematical Society* 60.6 (1954), pp. 503–515.

[6] James Bennett and Stan Lanning. "The netflix prize". In: *Proceedings of the 2007 KDD Cup and Workshop*. San Jose, CA, USA.

[7] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for Hyper-parameter Optimization". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. Granada, Spain, 2011, pp. 2546–2554.

[8] Dimitri P Bertsekas. *Dynamic programming and optimal control*. Vol. 1. Athena Scientific Belmont, Massachusetts, 1996.

[9] L. Bottou, F. E. Curtis, and J. Nocedal. "Optimization Methods for Large-Scale Machine Learning". In: *ArXiv e-prints* (2016).

[10] Leo Breiman. "Bagging predictors". In: *Machine Learning*. Vol. 24. 2. Springer. 1996, pp. 123–140.

[11] Leo Breiman. *Classification and regression trees*. Routledge, 1984.

[12] Leo Breiman. "Random Forests". In: *Machine Learning*. Vol. 45. 1. Springer. 2001, pp. 5–32.

[13] Thomas M Breuel. "Benchmarking of LSTM networks". In: *arXiv preprint arXiv:1508.02774* (2015).

[14] Eric Brochu, Vlad M. Cora, and Nando de Freitas. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning". In: *CoRR* abs/1012.2599 (2010).

[15] B. Ciciani, D. Didona, P. Di Sanzo, R. Palmieri, S. Peluso, F. Quaglia, and P. Romano. "Automated Workload Characterization in Cloud-based Transactional Data Grids". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum(IPDPSW)*. Vol. 00. 2012, pp. 1525–1533.

[16] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. “Deep Big Multilayer Perceptrons for Digit Recognition”. In: *Neural Networks: Tricks of the Trade: Second Edition*. Springer Berlin Heidelberg, 2012, pp. 581–598.

[17] R Dennis Cook and Christopher J Nachtrheim. “A comparison of algorithms for constructing exact D-optimal designs”. In: *Technometrics*. Vol. 22. 3. Taylor & Francis, Ltd., American Statistical Association, American Society for Quality. 1980, pp. 315–324.

[18] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues. “Chasing the Optimum in Replicated In-Memory Transactional Platforms via Protocol Adaptation”. In: *IEEE Transactions on Parallel and Distributed Systems* 26.11 (2015), pp. 2942–2955.

[19] D. D. Cox and S. John. “A statistical method for global optimization”. In: *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 2. 1992, pp. 1241–1246.

[20] Christina Delimitrou and Christos Kozyrakis. “HCloud: Resource-Efficient Provisioning in Shared Cloud Systems”. In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 2016, pp. 473–488.

[21] Christina Delimitrou and Christos Kozyrakis. “Quasar: Resource-efficient and QoS-aware Cluster Management”. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Salt Lake City, UT, USA, 2014, pp. 127–144.

[22] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. “Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters”. In: *Proceedings of the 6th ACM Symposium on Cloud Computing*. Kohala Coast, Hawaii, 2015, pp. 97–110.

[23] Li Deng. “The MNIST database of handwritten digit images for machine learning research [best of the web]”. In: *IEEE Signal Processing Magazine*. Vol. 29. 6. IEEE. 2012, pp. 141–142.

[24] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. “ProteusTM: Abstraction Meets Performance in Transactional Memory”. In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 2016, pp. 757–771.

[25] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. “Tuning database configuration parameters with iTuned”. In: *Proceedings of the VLDB Endowment*. Vol. 2. 1. VLDB Endowment. 2009, pp. 1246–1257.

[26] Michael D Ekstrand, John T Riedl, and Joseph A Konstan. “Collaborative filtering recommender systems”. In: *Foundations and Trends in Human–Computer Interaction*. Vol. 4. 2. Now Publishers, Inc. 2011, pp. 81–173.

[27] Ronald Aylmer Fisher. “Theory of statistical estimation”. In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 22. 5. Cambridge University Press. 1925, pp. 700–725.

[28] Yarin Gal, Yutian Chen, and Zoubin Ghahramani. “Latent Gaussian Processes for Distribution Estimation of Multivariate”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. Lille, France, 2015.

[29] Jacob R. Gardner, Matt J. Kusner, Zhixiang Xu, Kilian Q. Weinberger, and John P. Cunningham. “Bayesian Optimization with Inequality Constraints”. In: *Proceedings of the 31st International Conference on Machine Learning*. Beijing, China, 2014, pp. 937–945.

[30] Carlos A Gomez-Uribe and Neil Hunt. “The netflix recommender system: Algorithms, business value, and innovation”. In: *ACM Transactions on Management Information Systems*. Vol. 6. 4. 2016.

[31] Google. *Google Compute Engine*. https://cloud.google.com/compute/.

[32] Google Compute Engine. *Custom Machine Types*. https://cloud.google.com/compute/docs/machine-types#custom_machine_types.

[33] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. "The WEKA Data Mining Software: An Update". In: *ACM SIGKDD explorations newsletter*. Vol. 11. 1. 2009, pp. 10–18.

[34] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. "Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets". In: *Proceedings of the 12th European Conference on Computer Systems*. Belgrade, Serbia, 2017, pp. 589–604.

[35] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent W Freeh. "Scout: An Experienced Guide to Find the Best Cloud Configuration". In: *arXiv preprint arXiv:1803.01296* (2018).

[36] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential Model-based Optimization for General Algorithm Configuration". In: *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*. Rome, Italy, 2011, pp. 507–523.

[37] Donald R. Jones. "A Taxonomy of Global Optimization Methods Based on Response Surfaces". In: *Journal of Global Optimization*. Vol. 21. 4. Springer. 2001, pp. 345–383.

[38] Donald R. Jones, Matthias Schonlau, and William J. Welch. "Efficient Global Optimization of Expensive Black-Box Functions". In: *Journal of Global Optimization*. Vol. 13. 4. Springer. 1998, pp. 455–492.

[39] J. Kiefer and J. Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function". In: *The Annals of Mathematical Statistics* 23.3 (1952), pp. 462–466.

[40] Younghoon Kim and Kyuseok Shim. "Twitobi: A recommendation system for twitter using probabilistic modeling". In: *Proceedings of the IEEE 11th International Conference on Data Mining*. Vancouver, Canada, 2011, pp. 340–349.

[41] H. J. Kushner. "A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise". In: *Journal of Basic Engineering*. Vol. 86. 1. The American Society of Mechanical Engineers. 1964, p. 97.

[42] Remi R. Lam, Karen E. Willcox, and David H. Wolpert. "Bayesian Optimization with a Finite Budget: An Approximate Dynamic Programming Approach". In: *Proceedings of the 29th Neural Information Processing Systems Conference*. Barcelona, Spain, 2016, pp. 883–891.

[43] Remi Lam and Karen Willcox. "Lookahead Bayesian Optimization with Inequality Constraints". In: *Proceedings of the 30th Neural Information Processing Systems Conference*. Long Beach, CA, USA, 2017, pp. 1890–1900.

[44] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. Vol. 86. 11. IEEE. 1998, pp. 2278–2324.

[45] G. Linden, B. Smith, and J. York. "Amazon.com recommendations: item-to-item collaborative filtering". In: *IEEE Internet Computing* 7.1 (2003), pp. 76–80.

[46] Qing Liu and Donald A Pierce. "A note on Gauss—Hermite quadrature". In: *Biometrika*. Vol. 81. 3. Oxford University Press, Biometrika Trust. 1994, pp. 624–629.

[47] M. D. McKay, R. J. Beckman, and W. J. Conover. “A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code”. In: *Technometrics*. Vol. 21. 2. Taylor & Francis, Ltd., American Statistical Association, American Society for Quality. 1979, pp. 239–245.

[48] Microsoft Azure. *Virtual Machines*. https://azure.microsoft.com/en-us/services/virtual-machines/.

[49] John Mingers. “An empirical comparison of selection measures for decision-tree induction”. In: *Machine learning*. Vol. 3. 4. Springer. 1989, pp. 319–342.

[50] J Mockus, Vytautas Tiesis, and Antanas Zilinskas. “The Application of Bayesian Methods for Seeking the Extremum”. In: *Toward Global Optimization*. Vol. 2. Elsevier. 1978, pp. 117–128.

[51] Martin Pelikan, David E. Goldberg, and Erick Cantú-Paz. “BOA: The Bayesian Optimization Algorithm”. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*. Vol. 1. Orlando, FL, USA, 1999, pp. 525–532.

[52] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. “Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters”. In: *Proceedings of the 13th EuroSys Conference*. Porto, Portugal, 2018, 3:1–3:14.

[53] Warren B Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*. Vol. 703. John Wiley & Sons, 2007.

[54] J. Ross Quinlan. “Induction of decision trees”. In: *Machine learning*. Vol. 1. 1. 1986, pp. 81–106.

[55] Laura Elena Raileanu and Kilian Stoffel. “Theoretical comparison between the gini index and information gain criteria”. In: *Annals of Mathematics and Artificial Intelligence*. Vol. 41. 1. Springer. 2004, pp. 77–93.

[56] Carl Edward Rasmussen and Malte Kuss. “Gaussian Processes in Reinforcement Learning”. In: *Proceedings of the 16th Neural Information Processing Systems Conference*. Whistler, British Columbia, Canada, 2003.

[57] Francesco Ricci, Lior Rokach, and Bracha Shapira. “Introduction to Recommender Systems Handbook”. In: *Recommender Systems Handbook*. Springer, 2011, pp. 1–35.

[58] Herbert Robbins and Sutton Monro. “A stochastic approximation method”. In: *The annals of mathematical statistics* (1951), pp. 400–407.

[59] Ryan P. Adams. *A Tutorial on Bayesian Optimization for Machine Learning*. https://www.iro.umontreal.ca/~bengioy/cifar/NCAP2014-summerschool/slides/Ryan_adams_140814_bayesopt_ncap.pdf.

[60] J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. “Collaborative Filtering Recommender Systems”. In: *The Adaptive Web: Methods and Strategies of Web Personalization*. Springer Berlin Heidelberg, 2007, pp. 291–324.

[61] Samuel David Silvey. *Optimal design: an introduction to the theory for parameter estimation*. Vol. 1. Springer Science & Business Media, 2013.

[62] Michael Stein. “Large Sample Properties of Simulations Using Latin Hypercube Sampling”. In: *Technometrics*. Vol. 29. 2. Taylor & Francis, Ltd., American Statistical Association, American Society for Quality. 1987, pp. 143–151.

[63] Stephen M. Stigler. “Optimal Experimental Design for Polynomial Regression”. In: *Journal of the American Statistical Association*. Vol. 66. 334. American Statistical Association, Taylor & Francis, Ltd. 1971, pp. 311–318.

[64] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[65] Aimo Torn and Antanas Zilinskas. *Global Optimization*. Springer-Verlag, 1989.

[66] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. “Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics”. In: *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*. Santa Clara, CA, USA, 2016, pp. 363–378.

[67] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. “Large-scale Cluster Management at Google with Borg”. In: *Proceedings of the 10th European Conference on Computer Systems*. Bordeaux, France, 2015, 18:1–18:17.

[68] Christopher KI Williams and Carl Edward Rasmussen. “Gaussian processes for regression”. In: *Advances in neural information processing systems*. 1996, pp. 514–520.

[69] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. “Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach”. In: *Proceedings of the 8th ACM Symposium on Cloud Computing*. Santa Clara, CA, USA, 2017, pp. 452–465.