UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

Automated Planning to Support the Deployment and Management of Applications in Cloud Environments

Richard Joaquin Gil Martinez

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues
Co-Supervisors: Doctor Maria Antónia Bacelar da Costa Lopes
Doctor Peter Van Roy

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering
Jury final classification: Pass with Distinction

2019

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

Automated Planning to Support the Deployment and Management of Applications in Cloud Environments

Richard Joaquin Gil Martinez

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues
Co-Supervisors: Doctor Maria Antónia Bacelar da Costa Lopes
Doctor Peter Van Roy

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering
Jury final classification: Pass with Distinction

Jury

Chairperson: Doctor José Carlos Alves Pereira Monteiro, Instituto Superior Técnico, Universidade de Lisboa
Members of the Committee:
Doctor Luís Eduardo Teixeira Rodrigues, Instituto Superior Técnico, Universidade de Lisboa
Doctor António Manuel Ferreira Rito da Silva, Instituto Superior Técnico, Universidade de Lisboa
Doctor Ahmad Al-Shishtawy, RISE Research Institutes of Sweden, Sweden
Doctor Bert Lagaisse, KULeuven, Belgium

Funding Institutions
European Commission EACEA
Fundação para a Ciência e Tecnologia

2019

Acknowledgement

The work described in these pages would not have been possible without the support of many people, to whom I am in debt.

First of all, I have to thank my supervisor and co-supervisors: Prof. Luís Rodrigues, for providing the conditions and the environment to explore new ideas and for helping me keep the big picture in mind; Prof. Antónia Lopes, for her thoroughness and dedication, which helped me clarify theoretical principles and concepts guiding my research; and Prof. Peter Van Roy, for the discussion of interesting ideas. I also have to thank co-authors Zhongmiao Li and Francisco Duarte, for the opportunity to collaborate and learn from them. Working with them gave me the opportunity to broaden the spectrum of my research and helped me grow as a scientist.

I would like to thank the faculty and students of the Distributed Systems Group (GSD) of INESC-ID, for creating an environment where to share ideas and experiences openly, and for the many times their feedback helped me improve my research. In no particular order (and hopping not to forget anyone), my sincere thanks to: António Rito Silva, Paolo Romano, Miguel Correia, Rodrigo Rodrigues, Ines Lynce, Vasco Manquinho, Nuno Santos, Igor Zavalyshyn, Daniel Porto, João Loff, Zhongmiao Li, Paolo Laffranchini, Manuel Bravo, João Neto, Amin Mohtasham, Nancy Estrada, David Gureya, Shady Issa, Diogo Barradas, Rodrigo Bruno. Also, a big thanks to the administrative personnel, who were always helpful and many times brighten my days, in particular: Paola Barrancos and Vanda Fidalgo.

I would like to thank all the professors, students, and administrative personnel from the institutions participating in the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC): Instituto Superior Técnico (IST), Université Catholique de Louvain (UCL), Universitat Politècnica de Catalunya (UPC), and Royal Institute of Technology (KTH). Thank you for the efforts in organizing workshops to debate research goals and ideas and for handling all the bureaucratic aspects. A special word of appreciation to Prof. Leandro Navarro, Prof. Luís Rodrigues, and Prof. Peter Van Roy. A word of appreciation to Ana Barbosa and Vanessa

Maons, for helping me with all administrative procedures at IST and UCL.

A special thank you to my friends (listing them all would be impossible), who supported me, helped me, and were patient enough to stay by my side. To all those who were faithful to our friendship during hard times and had a word of encouragement for me. Also, thanks to all the people of the christian communities in Vineyard Church Brussels and Hillsong Portugal, for their friendship and support.

The biggest thanks to my family. My father, Richard Gil, who motivated me to pursue my doctoral studies and tirelessly pushed me throughout the process. My mother, Marlene Martinez, for her prayers, for the long conversations and for caring about me. My sister, Daniela Gil, for giving me a wake up call whenever I lost focus and for staying close. And my sister, Deris Gil, who supported me in all the ways a sister can support a brother. To all my extended family, who always had words of admiration, love, and encouragement. To all the Venezuelan people who held their heads high during hardship.

A final word of appreciation to the European Commission EACEA and to the Fundação para a Ciência e Tecnologia (FCT) in Portugal, for their financial support. The work presented in the thesis was supported by:

- European Commission EACEA via a full doctoral scholarship for the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC), funded under project reference FPA 2012-0030.
- Fundação para a Ciência e Tecnologia (FCT), via research scholarships under projects with reference UID/CEC/50021/2013, CMUP-EPB/TIC/0042/2013, and PTDC/EEI-SCR/1741/2014

To all those who contributed to the
work described in these pages, and
to all those who will read this in the
future and find it useful.

Abstract

Cloud computing has enabled myriad of computer applications to benefit from dynamic provisioning of resources. Computer applications must often adjust their resources in response to changes in the environment, in order to satisfy business-defined goals. However, engineering decision-making mechanisms to help the deployment and management of resources in cloud contexts is a challenging task. In fact, applications running on cloud infrastructures call for automated mechanisms that: (1) explore efficiently large solution spaces (defined by the combination of machine types, provisioning actions, and state transitions expected in the temporal horizon); (2) generate deliberate plans to operate the system in a way that satisfies requirements, maximizes performance and minimizes operational costs; and (3) support the definition and revision of policies to adapt the system under expected conditions. *Automated Planning*, the area of artificial intelligence concerned with synthesizing plans of actions to achieve a goal, offers opportunities to address these challenges.

This thesis focuses on the design and evaluation of mechanisms that exploit *automated planning to support the deployment and management of applications running in cloud environments*. To this purpose, this thesis presents three contributions: (1) a solution to the (offline) generation of reactive policies that exploits classical planning languages and tools to support the definition and revision of policies, applicable under common conditions; (2) a solution to the (online) generation of proactive plans that takes advantage of temporal planning and behavioral predictions to reconfigure interactive applications; and (3) a solution to the (offline) generation of execution policies that resorts to probabilistic planning to deal with the uncertainty caused by spot instance revocations in the deployment of workflow applications. These proposals have been evaluated using realistic case studies of elastic scaling and workflow executions in the cloud.

Results support the claim that automated decision-making mechanisms that rely on planning are scalable and responsive, and able to guide the system to satisfy requirements, optimize performance and minimize operational costs.

Resumo

A computação em nuvem permitiu que várias aplicações se beneficiassem da reserva dinâmica de recursos. Essas aplicações necessitam de ajustar seus recursos em resposta a mudanças no ambiente, a fim de satisfazer metas de qualidade de serviço definidas pelo negócio. No entanto, a concepção dos mecanismos de tomada de decisão usados para gerir a reserva e a gestão de recursos em contextos de computação na nuvem é uma tarefa complexa. Na verdade, as aplicaçães na nuvem exigem mecanismos automáticos para: (1) explorar eficientemente grandes espaços de solução (definidos pela combinação dos tipos de máquina, das ações de reserva e das transições de estado esperadas no horizonte temporal); (2) gerar planos para operar o sistema de uma maneira que satisfaça os requisitos, maximize o desempenho e minimize os custos operacionais; e (3) apoiar a definição e revisão de políticas para adaptar o sistema nas condições esperadas. O *Planeamento Automático*, a área de inteligência artificial dedicada a sintetizar sequências de ações para atingir um objetivo, permite enfrentar esses desafios.

Esta tese aborda o desenho e avaliação de técnicas que exploram o *planeamento automático para suportar a reserva e a gestão de aplicações executadas na nuvem.* Com este objectivo, a tese apresenta três contribuições: (1) uma solução para a geração (offline) de políticas reativas, que explora linguagens e ferramentas de planeamento temporal para apoiar a definição e revisão de políticas, aplicáveis sob condições comuns; (2) uma solução para a geração (online) de planos proativos, que aproveita o planeamento temporal de longo prazo e as previsões comportamentais para reconfigurar aplicações interativas; e (3) uma solução para a geração (offline) de políticas de execução, que recorre ao planeamento probabilístico para lidar com a incerteza causada por revogações de instâncias efémeras (*spot instances*, em inglês) na concretização de aplicações baseadas em fluxos de trabalho. Essas propostas foram avaliadas usando casos de estudo realistas que ilustram a escalabilidade elástica e execuções de fluxo de trabalho na nuvem.

Os resultados suportam a hipótese de que os mecanismos automáticos de tomada de decisão baseados em planeamento são escaláveis, ágeis, e capazes de orientar o sistema para satisfazer os requisitos do utilizador, otimizar o desempenho e minimizar os custos operacionais.

Keywords
Palavras Chave

Keywords

Cloud Computing

System Management

Adaptive Systems

Decision Making

Planning

Palavras Chave

Computação em Nuvem

Gestão de Sistemas

Sistemas Adaptativos

Tomada de Decisões

Planeamento

Index

List of Figures

List of Tables

1 Introduction

Cloud computing represents a global market of over $186.4 billion, of which the fastest-growing segment is the infrastructure services, projected to grow 35.9% in 2019 (Gartner 2018). Cloud computing has enabled myriad of computer applications to benefit from dynamic provisioning of resources. It offers desirable features such as: *virtualization*, the instantiation of virtual machines with pre-defined sizes decoupled from the physical infrastructure where they are deployed; *provisioning*, the opportunity to rent and allocate resources on-demand; and a *pay-as-you-go* economic model, such that users only pay for the actual resources consumed. The combination of these features offers opportunities for computer systems to become more scalable, cost-efficient, and autonomous.

One example of the services that make cloud computing attractive is *elastic scaling*. Elastic scaling enables cloud applications to dynamically accommodate resources in a pool of virtual servers, in response to variations in their workload. By doing so, cloud applications can avoid unnecessary costs in their operations, caused by capacity over-provisioning. To manage elastic scaling, applications can put in place a set of policies to regulate the activation and termination of machines in their server pool. For instance, a reactive policy may dictate that a new virtual machine must be activated when the average CPU utilization reaches a given threshold, defined by the system expert. Indeed, most cloud providers support elastic scaling through the definition of reactive rules (e.g. Amazon Auto-Scaling in Amazon EC2[1]). Yet, these rules are restrictive in what they express and do not fully exploit the flexibility of the cloud infrastructure.

Cloud providers offer a wide range of virtual machines for applications with different requirements (e.g. general purpose, compute optimized, memory optimized), with different "sizes"

[1] https://aws.amazon.com/

pre-defined by the cloud provider (e.g. small, medium, large) that are allocated distinct amounts of computational resources (e.g. number of CPU cores, memory size, etc.). Virtual machines can also be rented in different price markets, as on-demand instances or revocable instances (a.k.a. spot instances). On-demand instances are billed a fixed price per time unit (e.g. by the hour), with reliability guarantees to the user. Revocable instances have dynamic prices that vary according to a bidding market and can be revoked at any time by the cloud provider, when the price of the instance exceeds the bid placed by the user. Ideally, cloud applications can take advantage of the diversity of machine sizes and market prices to define policies that optimize their performance and guarantee cost-effective operations.

However, defining policies for dynamic provisioning of resources in the cloud is a challenging task. First, a wide variety of machine sizes can be combined together into many possible configurations. These machines can also be activated or terminated in different orders to reach the desired configuration. Deciding on the best configuration and the order of the provisioning actions that satisfies the system requirements, optimizes its performance, and minimizes its operational costs, is complex. Second, cloud infrastructures are affected by uncertainty. Uncertainty due to performance variability and interference makes so that applications deployed in two different machines of the same type perform differently. Uncertainty provoked by the unexpected revocation of spot instances can also affect the efficiency and reliability of the application. Thus, applications must make decisions that account for actions that may have multiple outcomes and (sometimes undesirable) effects on the system. Third, reaction times are limited, scaling actions are not instantaneous and cloud resources are rented in time units (e.g. by the hour). So, in order to make effective and pertinent decisions, applications must consider temporal factors, such as server activation times or the expected behavior of the environments in a time horizon. Finally, applications must translate high-level business objectives into operational goals and ensure that policies do not violate system constraints and user preferences.

Policies are not only hard to define, but also difficult to maintain. Policies are usually defined in the context of stable cloud computing services, according to specific business objectives. Still, cloud contexts are subject to constant changes (e.g. the introduction of new hardware can make so that new machine types are available to the application). Also, business objectives and system requirements are constantly refined as the system evolves. In consequence, policies must be updated frequently.

The complexity of decision-making for the definition of policies that govern the deployment and management of computer applications running in cloud environments may be an impairment to their further development. Currently, cloud applications can enable low-level corrective policies to react to environmental changes, usually expert-defined event-condition-action rules (as discussed previously). Still, more elaborate (high-level) deliberate plans are needed to guide the system under uncertainty towards the desired configuration. Yet, plans as such are hard to define by experts.

Certainly, cloud applications are in need of automated mechanisms for decision-making that: (1) facilitate the exploration of large solution spaces (defined by the combination of machine sizes, the provisioning actions, and the state transitions expected in the temporal horizon); (2) generate deliberate plans to operate the system in a way that satisfies requirements, maximizes performance and minimizes operational costs; and (3) support the automated definition and revision of policies that adapt the system under common conditions and that are easily updated in changing business contexts. In this, the problem of deploying and managing applications in cloud environments is akin to that of autonomic computing, given the requirements for dynamic accommodation of resources, on-the-fly tuning of parameters, robustness to environmental changes, and adjustability to variable user requirements.

Autonomic computing aims at reducing the barrier that complexity poses to further development of computer systems by integrating intelligence into the management process (Kephart & Chess 2003). By closing the operational loop via an external control mechanism, it is possible to monitor the performance of the system, analyze its behavior, and, when necessary, plan and execute proper adaptation actions to guide the system towards business-defined goals. Systems that are able to autonomously adapt under changing conditions are known as *self-adaptive*.

Planning, the decision-making task of finding a course of action from an initial state to a desired state that satisfies the requirements of the system and meets the operational goals, is a fundamental piece to enable self-adaptation. Planning benefits from having an broad view on how actions can be combined together to sort out the environmental changes and lead the system to meet its requirements, while optimizing performance and minimizing costs. Automated planning, the area of artificial intelligence concerned with synthesizing plans of actions to achieve a goal, is rich in languages, algorithms, and tools that could potentially improve the deployment and management of computer system with complex decision-making problems.

To define a planning problem it is required some knowledge about the system configuration, the adaptation actions and their effects, and the user preferences and business goals. At its simplest form, the planning problem model defines: (1) all valid configurations of the system, (2) the adaptation actions that can be applied by the system with their expected effects in the system configuration, (3) the performance indicators that serve to guide the adaptation, and (4) the goals that define the desired configurations. Depending on other factors that affect decision-making, the planning problem can be defined as classical, probabilistic or temporal. The *classical planning* problem explores the space of possible system states to decide a deterministic plan that leads the system to the target state, which satisfies the goals. Additionally, when non-determinism can be captured in the form of probabilistic effects, *probabilistic planning* can resolve models where actions can lead the system to different possible states and find the (probabilistic) plan that optimizes the system performance and meets the goals. What is more, to take in consideration the temporal factors that may affect the adaptation, *temporal planning* can tackle problems characterized by actions with variable duration and concurrency, treating time as a continuous exogenous variable or using augmented state variables that include time.

To engineer an adequate planning solution, one must not only consider the complexity captured in the planning problem definition, but also the time available for planning. For instance, in the case of elastic scaling, applications exposed to steady workload variations typically have enough time to react, while applications subject to faster workload changes usually do not. Reactive planning online risks failing at finding a timely solution when the planning problem is complex and reaction times are tight. As an alternative, offline planning can take advantage of longer search times to find adequate solutions that can later be encapsulated into policies to manage operations at run-time.

1.1 Problem Statement

Previous experiences offer evidence that autonomic computing may alleviate the management burden for applications running in cloud environments (Brun, Serugendo, Giese, Kienle, Litoiu, Müller, Pezzè, & Shaw 2009; Krupitzer, Roth, VanSyckel, Schiele, & Becker 2015). Nevertheless, cloud applications are still unable to exploit the power of intelligent planning fully (Salehie & Tahvildari 2009; Weyns 2019). In particular, the mechanisms that guide the deployment and management of applications in the cloud are still rather primitive and could benefit from automated planning. Considering this context, the thesis addresses the following question:

How can planning techniques be engineered to automate the deployment and management of computer applications running in cloud environments?

To make steps towards answering this question, this thesis addresses the use of automated planning for the adaptation of cloud applications in three different scenarios, as discussed below.

1.1.1 Scenario 1

Reactive policies used to manage resources in the cloud have several limitations. First, reactive policies are usually defined by operators and system managers. Sadly, reactive policies defined solely by humans are usually limited in the way they conduct reconfiguration and overly simplistic on how actions are combined. Second, reactive policies do not exploit the benefits of deliberate decision-making. Most policies, defined in the form of event-condition-action rules, are executed greedily and can lead the system to states that are locally optimal at best. Third, reactive policies are difficult to maintain in business contexts subject to frequent changes, such as cloud environments. These policies require to be constantly updated to: (1) account for new machine types and prices made available; (2) consider new system requirements and dependability constraints; and (3) adjust to changing business goals and user preferences. Ideally, planning can support the automated definition and revision of policies that describe how to adapt the system under common conditions and that can be easily updated in changing business contexts.

In this scenario, the thesis addresses the following question:

How can classical planning be used for the (offline) generation of policies that support elastic scaling of interactive applications in cloud environments?

1.1.2 Scenario 2

Many interactive applications running in cloud environments demand some reconfiguration immediately after accommodating their resources. Typically, data reconfigurations have some negative impact on the performance of the system; thus, applications attempt to minimize the number of reconfiguration actions and their impact. Additionally, these applications are often exposed to fast-changing environments. Therefore, adaptation actions must be enforced even before changes in the environment take place. This is to avoid that data reconfigurations occur during a period of high stress that could further degrade the performance. Finally, cloud providers rent resources in time units and, when billing periods are long (e.g. per hour), decisions can be better made having a long-term view of the environmental changes in the horizon. Ideally, interactive applications running in cloud environments can benefit from controllers that: (1) decide adaptation action proactively by anticipating environmental changes; (2) consider long-term decisions that account for the billing periods of the cloud provider; and (3) take in consideration the impact of reconfiguration in the system performance when making decisions.

In this scenario, the thesis addresses the following question:

How can temporal planning be used for the proactive reconfiguration (online) of interactive applications in cloud environments?

1.1.3 Scenario 3

Many cloud providers offer the possibility of renting transient resources (a.k.a. *spot instances*). Cloud applications may take advantage of transient resources, provided that potential failures provoked by the revocation of these machines do not affect the application services given to the customers. For instance, muti-tier web or data processing applications could benefit of discounted costs by being deployed dynamically in the cloud, using a mix of on-demand and spot instances. These applications are usually represented as a workflow of tasks with timely requirements. Ideally, planning can be used to define policies for the dynamic deployment of workflows in the cloud that guarantee timely executions and reliability at minimal costs.

In this scenario, the thesis addresses the following question:

How can probabilistic planning be used for the (offline) generation of policies for the execution of workflows using spot instances in cloud environments?

1.2 Summary of Contributions

This thesis explores the three opportunities for the use of automated planning to support the deployment and management of applications in cloud environments, identified above. These opportunities are materialized in the following contributions:

Planning to Support Reactive Policies for Elastic Scaling: This contribution discusses how to integrate AI planning into the generation of high-level adaptation policies that are: (1) easily derived from the system models and human expertise; (2) effective at guiding adaptation towards the best reachable system state; and (3) able to comply to dependability constraints imposed to the system. Automated planning is used as a complement to human decision-making. A standard planning language is used to encode a temporal planning problem, while planning tools help decide the target configuration and the adaptation plan towards the target configuration. In addition, a novel scanning mechanism is proposed to build policies that cover common system conditions. The solution is evaluated in the context of policy revision due to new dependability constraints introduced by the system experts.

Planning to Support Proactive Scaling and Reconfigurations: This contribution explores the reconfiguration of cloud-enabled applications using controllers that combine proactive techniques with the ability to manage heterogeneous resources. The solution relies on temporal patterns obtained from historical data to predict the evolution of the workload and to initiate adaptation before the service quality is affected. Decisions are made based on knowledge about the workload curve, the cloud resources, and the initial system configuration. The proposed solution is materialized in Augure, a controller that uses linear programming at run-time to search the space of actions in long-term horizons and to select a plan that: (1) minimizes the price billed by the cloud provider and (2) mitigates the negative side-effects of reconfiguration on the service quality. In addition, to study proactive controllers that make (greedy) decisions looking at short-term horizons, this contribution introduces Vadara+, an extension of Vadara (Loff & Garcia 2014) that manages heterogeneous resources. Augure is evaluated for the live reconfiguration of an online transaction processing database system, called E-Store (Taft, Mansour, Serafini, Duggan, Elmore, Aboulnaga, Pavlo, & Stonebraker 2014), and compared to other controllers: Reactive, Vadara, and Vadara+.

Planning to Support Workflow Executions with Spot Instances: This contribution studies the use of automated planning as a tool to optimize the execution of deadline-constraint workflows in cloud environments. The proposed solution derives policies using models that capture the non-deterministic effects of deployment actions, to account for the probability of revocation of a spot instance. Planning is used to solve automatically generated Markov Decision Processes (MDP) that model the execution of the workflow and to explore the state space using well-known algorithms that lead to the optimal policy. Static policies are derived offline, before the execution of a job. These policies are then executed at run-time and guide the selection of deployment actions, depending on the occurrence of failures and the actual task completion times. The solution is evaluated for the execution a real-world scientific workflow application, Epigenomics (Juve, Chervenak, Deelman, Bharathi, Mehta, & Vahi 2013). The solution is compared to two other scheduling frameworks that consider spot instances: LTO (Poola, Ramamohanarao, & Buyya 2014) and Dyna (Zhou, He, & Liu 2016).

1.3 Summary of Results

Considering the contributions listed above, the main results present in the thesis are the following:

- An implementation of an automated planning solution to support the definition and revision of reactive policies for elastic scaling in the cloud, and its evaluation via simulations in a case study.

- An implementation of an automated planning solution to support the proactive scaling and reconfiguration of cloud applications, and its evaluation via simulations and a prototype deployment in E-Store, an online transaction processing database system.

- An implementation of an automated planning solution to support workflow executions using spot instances in the cloud, and its evaluation via simulations with Epigenomics, a scientific workflow application.

1.4 Thesis Structure

The remaining of the thesis has the following structure:

Chapter 2: introduces fundamental concepts which are relevant for the context of the contributions presented in the thesis;

Chapter 3: presents and evaluates an automated planning solution to support the definition and revision of reactive policies for elastic scaling in the cloud;

Chapter 4: presents and evaluates an automated planning solution to support the proactive scaling and reconfiguration of cloud applications;

Chapter 5: presents and evaluates an automated planning solution to support workflow executions using spot instances in the cloud;

Chapter 6: concludes the thesis summarizing the results derived from the thesis and discussing pointers for future work.

2 Fundamental Concepts and State of the Art

2.1 Cloud Computing and Applications

2.1.1 Cloud Computing

Cloud computing has emerged as a model that changes the way software applications are deployed and managed. Cloud computing offers three attractive features: (1) *virtualization*, the instantiation of virtual servers with pre-defined computational capacities decoupled from the actual physical infrastructure where they are deployed; (2) *provisioning*, the flexibility to scale a pool of computational resources by activating or terminating servers on-the-fly; and (3) a *pay-as-you-go* economic model, the chance to pay for the consumed resources by time units.

Cloud providers offer a wide variety of virtual machines customized for different applications: general purpose, computational intensive, memory intensive, etc. For each type of application, there exists a wide selection of server sizes with pre-defined configurations of CPU, memory, storage, and network capacity. For instance, a computationally optimized machine of extra large size *c5.xlarge* in Amazon EC2 has 4 CPUs and 8 GiB of memory. Currently, Amazon offers more than six machine sizes for each application type[1].

Cloud providers typically follow a linear pricing scheme, i.e. prices increase linearly with the size of virtual machines of the same type. For instance, in Amazon EC2, computationally optimized machines of extra large size *c5.xlarge* have a price of 0.17$/hour, while machines with double the resources *c5.2xlarge* have a price of 0.34$/hour[2]. Cloud providers charge these costs to the application, depending on the amount of time consumed by a rented virtual machine instance. Usually, applications are charged by the hour, meaning that the time consumed by the application is counted in hours and any extra time counts as a complete extra hour. That is, any spare time is payed by the application even if the server remains idle. More recently,

[1] https://aws.amazon.com/ec2/instance-types/
[2] https://aws.amazon.com/ec2/pricing/on-demand/

some cloud providers have introduced a billing scheme in seconds, which gives more flexibility to the application and potentially reduces the cost generated by instances in idle state.

Cloud providers also offer revocable instances (*spot instances*[3] in Amazon EC2 or *preemptible instances*[4] in Google GCE). Revocable instances are machines that take advantage of spare compute capacity in the cloud, available to the user at steep discounts compared to on-demand prices, but that can be interrupted by the cloud provider when the capacity is needed. These instances are regulated by a bidding market, such that the instance is granted to the highest bidder (and revoked when another user places a higher bid) and the cost is decided by the cloud provider dynamically, based on the market demand[5]. Revocable instances can help reduce operational costs of applications, since they are rented at a fraction of the cost of on-demand instances. Yet, since they can also be evicted at any time, they can introduce uncertainty about the composition of the server pool and its performance.

2.1.2 Elastic Scaling of Interactive Applications

Interactive applications naturally operate in changing environment, where user demand is variable over time. These applications can benefit from the dynamic reservation of resources in response to variations in their workload, an ability known as *elastic scaling*. Cloud computing has enabled many interactive applications to benefit from elastic scaling, providing easy access to virtual machine types that users can request and release as needed.

Applications in the cloud can reserve resources through two main *scaling actions*: (1) to *activate* a server, increasing the computational capacity of a pool of servers; and (2) to *terminate* a server, decreasing the computational capacity of a pool of servers. Naturally, these scaling actions have associated costs. Indeed, applications pay a fixed price (per time unit) for servers activated on-demand. The business objectives that regulate the elastic scaling of resources in the cloud can be often classified in three groups: (1) to meet the requirements of the system, e.g. in terms of dependability constraints; (2) to optimize the performance of the application (e.g. to keep the response time experience by the user under a given threshold); and (3) to minimize the monetary costs associated to the rental of resources.

[3] https://aws.amazon.com/ec2/spot/
[4] https://cloud.google.com/preemptible-vms/
[5] https://aws.amazon.com/ec2/spot/

Challenges of Elastic Scaling: Elastic scaling in the cloud is a hard task, mainly due to the presence of heterogeneous resources, performance variability, the impact of reconfigurations, and the limited reaction times. In the following, these factors are discussed:

1. *Managing heterogeneous resources:* An important dimension of elastic scaling is the ability to accommodate virtual machines of different sizes and prices. Ideally, cloud applications can exploit the diversity of machine types to potentially achieve a better fitting of the computational capacity to the workload. Managing heterogeneous resources is difficult, though. The best composition of the server pool can be a mix of different machines types. Finding the right composition is often complex and time-consuming. Also, scaling the pool towards the best composition may require activating and terminating servers of different types in a specific order, demanding that the controller is able to explore an even larger space of scaling actions to find the best plan.

2. *Performance variability:* While applications can decide which virtual machine to activate, the underlying physical infrastructure remains hidden. So, applications have little control over "where" their virtual machines are deployed. This is relevant because cloud services are known to suffer from significant performance variability and unpredictability (Leitner & Cito 2016; Delimitrou & Kozyrakis 2013). This is often caused by: (1) hardware heterogeneity, if virtual machines are instantiated in physical machines with different hardware features; (2) resource contention, when virtual machines col-located in the same physical machine contend for limited resources, e.g. processing power; and (3) failures, usually caused by hardware deterioration, software updates, or system malfunctioning. These factors impact the performance of the application differently depending on the application type and the infrastructure of the provider. Sadly, due to the unobservability and uncontrollabilty of these factors, applications can hardly predict the behavior of newly deployed virtual machines. Therefore, controllers must be able to handle uncertainty, e.g. by considering that the execution of scaling actions may lead to multiple possible outcomes.

3. *Reconfiguration impacts:* Scaling actions can penalize the system performance while being executed. This is relevant in stateful applications, where scaling actions may require data to be transferred among replicas. The state transfer not only consumes network resources, but also processing capacity in active replicas, contributing to increased latency and reduced throughput. For instance, storage database systems usually perform costly

migration procedures for re-balancing after scaling actions (Lim, Babu, & Chase 2010; Taft, Mansour, Serafini, Duggan, Elmore, Aboulnaga, Pavlo, & Stonebraker 2014). Controllers must take into account reconfiguration impacts when making decisions to scale resources, in order to mitigate their negative effects on performance and availability.

4. *Limited reaction times:* Elastic scaling in public clouds is mostly carried out using reactive rules. Cloud applications can set reactive policies to automatically scale resources when a performance indicator reaches a threshold (e.g. to activate a server when CPU utilization is above 80%). Yet, monitoring times and long server activation times slow down the ability to react to sudden changes. In scenarios subject to sudden changes, reactive scaling may cause delayed or dropped requests that ultimately cause profit loss.

2.1.2.1 Related Work

Due to its many benefits and practical relevance, elastic scaling of application in cloud computing has deserved significant attention in the literature. In the following, a summary of the most significant contributions to the state-of-the-art is presented.

2.1.2.1.1 Reactive Scaling Techniques A vast majority of the commercial and academic approaches resort to reactive policies to manage resources. The policies that govern reactive scaling exist in two forms: manual and automatic. A manual policy of elasticity means that the user is responsible for monitoring the virtual environment and applications and for executing all scaling actions. The cloud provider enables an interface (e.g an API) with which the user interacts with the system. An automatic policy means that the control and the actions are taken by the cloud system or by the application, in accordance with user-defined rules and settings, or specified in the Service Level Agreement (SLA). The controller uses monitoring to collect information like CPU load, memory and network traffic to decide when and how scale resources.

Reactive solutions are mostly based in event-condition-action rules. Each rule is composed of a set of conditions that, when satisfied, trigger scaling actions in the underlying cloud. Every condition considers an event or a metric of the system which is compared against a threshold. The information about metrics values and events is provided by the infrastructure monitoring system or by the application.

The use of reactive techniques is quite common and is found in most commercial providers: Amazon EC2 [6], Google GCE [7], Microsoft Azure [8], RackSpace [9], Rightscale [10], Scalr [11]. Example of academic works in which the resources are managed manually are Elastin (Neamtiu 2011) and Work Queue (Rajan, Canino, Izaguirre, & Thain 2011), and automatically, is TIDE (Meng, Liu, & Soundararajan 2010)

2.1.2.1.2 Proactive Scaling Techniques While commercial solutions are still dominated by reactive scaling policies, recent academic works have proposed predictive approaches that uses heuristics and mathematical/analytical techniques to anticipate the system load behavior, and based on these results, decide when and how to scale resources. Temporal predictors (e.g., time series with moving average, auto-regression, exponential smoothing, or neural networks) and non-temporal predictors (e.g., support vector machines or decision trees) are used to estimate future workload values and required resources, as discussed in many surveys (Qu, Calheiros, & Buyya 2018; Lorido-Botran, Miguel-Alonso, & Lozano 2014; Hummaida, Paton, & Sakellariou 2016). An empirical evaluation of most common techniques is presented in (Kim, Wang, Qi, & Humphrey 2016). Besides, some authors propose to combine several of these techniques to improve the accuracy of the prediction, e.g., to combine time-series forecasting using genetic algorithms (Messias, Estrella, Ehlers, Santana, Santana, & Reiff-Marganiec 2016).

Many proactive scaling techniques have been proposed in the literature. For brevity, only the most relevant to the thesis are presented here. PRESS (Gong, Gu, & Wilkes 2010) is a predictive (vertical) scaling engine, that leverages signal processing and statistical learning algorithms to achieve short-term predictions of resource requirements online. For workloads with repeating patterns, it derives a signature from historic resource usage patterns via a Fast Fourier Transform (FFT), and for applications without repeating patterns, it uses a Discrete-Time Markov Chain (DTMC) to build a short-term prediction of future metric values. CloudScale (Shen, Subbiah, Gu, & Wilkes 2011) is a system that automates fine-grained (vertical) scaling, based on time series of resource usage to predict the resource demands in the short-term. AGILE (Nguyen, Shen, Gu, Subbiah, & Wilkes 2013) adjusts dynamically the number of same-sized

[6] https://aws.amazon.com/
[7] https://cloud.google.com/compute/
[8] https://azure.microsoft.com/en-us/
[9] https://www.rackspace.com/
[10] https://www.rightscale.com/
[11] https://www.scalr.com/

virtual machines assigned to a cloud application, using wavelet transforms to provide a medium-term resource demand prediction with enough lead time to start up new server instances before performance falls short. Authors in (Jiang, Lu, Zhang, & Long 2013) propose an auto-scaling scheme that uses machine learning and time series historic data to predict the average number of web requests in the long-term future, and makes decisions to (horizontally) scale a pool of same-sized virtual machines based on that prediction. Vadara (Loff & Garcia 2014) is a generic elasticity framework that enables the use of pluggable cloud-provider-agnostic elastic strategies and uses a combination of predictive workload forecasting techniques to estimate the behavior of the workload to make short-term scaling decisions. InteliScaler (Shariffdeen, Munasinghe, Bhathiya, Bandara, & Bandara 2016) is a proactive auto-scaler for platform-as-a-service cloud, that combines workload prediction mechanism based on time-series forecasting with machine learning techniques to estimate the workload in the short-term, and make decisions taking into account the billing schemes of the provider (e.g. per hour).

2.1.2.1.3 Scaling and Data Reconfiguration Previous work has been dedicated to scaling resources for muti-tier applications that require some data reconfiguration. Those of particular interest are presented here. In (Lim, Babu, & Chase 2010), an elastic controller for stateful applications is proposed and tested with a popular distributed storage system. The controller makes scaling decisions considering the delay of actuators and the need to re-balance data across replicas. CloudScale (Shen, Subbiah, Gu, & Wilkes 2011) also automates elastic scaling and can resolve scaling conflicts between applications using migration. It employs a migration based conflict handling that estimates the penalties in Service Level Objectives (SLO) in resource provisioning and decides what virtual machine to migrate and when, considering the estimated penalties in performance. The Transactional Auto Scaler (TAS) presented in (Didona, Romano, Peluso, & Quaglia 2014) is a system for automating scaling of in-memory transactional data grids, e.g. NoSQL data stores or distributed transactional memories. It uses a performance forecasting methodology that combines analytical modeling and machine-learning, and manages heterogeneous resources. The SCADS Director (Trushkowsky, Bodík, Fox, Franklin, Jordan, & Patterson 2011) addresses the problem of scaling a distributed storage system under stringent performance requirements. It uses a performance model coupled with workload statistics to predict whether each server is likely to continue to meet its SLOs, and incorporates an estimation of the degradation due to data reconfiguration to decide when and what resources to scale.

2.1.2.1.4 Managing Heterogeneous Resources Most controllers in the literature focus on managing a pool of servers of the same size, thus simplifying the decision to how many servers to activate or terminate (Hummaida, Paton, & Sakellariou 2016; Galante & Bona 2012). Approaches that manage heterogeneous resources have been proposed before for resource provisioning and allocation, though. In (Srirama & Ostovar 2014) a controller for resource provisioning finds a cost-optimal setup of various machine sizes that satisfy the requirements of the incoming workload for workflow applications. A linear programming model is built, taking as input the workload of each task, the processing capacity, periodic costs and activation times of each machine type, and the age of running instances. The model, solved with constraint solvers, provides the optimal number of machines from each type that must be added to or removed. In (Wang, Gupta, & Urgaonkar 2016), authors go further assuming fine-grained scaling of CPU and memory 'within' an already procured virtual machine; a form of vertical scaling. The controller uses rather simplistic policies to make decisions on how much CPU and memory to add or remove from the instances. In (Verma, Gangadharan, Narendra, Ravi, Inamdar, Ramachandran, Calheiros, & Buyya 2016), a dynamic resource demand prediction and allocation framework is presented. This approach adds the service tenants to virtual machines that match the predicted load and allocates the virtual machines to physical machines using a best-fit decreasing heuristic. In (Ma, Zhang, Zhang, & Zhang 2016), genetic algorithms are used to solve a dynamic resource allocation and cost optimization model to find near-optimal solutions efficiently.

2.1.2.2 Limitations and Open Issues

Research in the last decade has proposed elaborate techniques to automate elastic scaling. However, there are still some limitations in their current ability to satisfy business goals in terms of performance optimization and cost minimization, as well as unexplored opportunities to exploit the flexibility offered by cloud computing.

1. *Reactive policies are restrictive.* Event-condition-action rules are insufficient to capture reasoning for complex decision-making and, thus, are unable to intelligently manage resource provisioning for interactive applications. Cloud applications can benefit from the definition of more complex policies that capture strategic reasoning and that implement sequential/parallel scaling actions. Current reactive policies are also unable to abstract from high-level business objective and user preferences. Cloud applications can benefit

from mechanism to translate high-level objectives and user preferences into operational policies. These mechanism should be flexible enough to allows the re-definition of policies may the business context evolve.

2. *Elastic scaling is reduced to a greedy execution of actions.* Reactive controllers employ a set of rules that are executed one-at-a-time, in a best-effort manner. Similarly, proactive controllers use workload predictions to make punctual decisions in the near future; these decisions are also executed greedily. Most proactive approaches are based on short-term predictions and, even those that predict the workload in the long term, do not exploit the full benefits of long-term planning (at best, they employ heuristics to allocate resources). Greedy scaling is undesirable; it may provoke unnecessary oscillations and lead to sub-optimal solutions, unable to lead the system to its optimal configuration in a way that minimizes long-term cost. Cloud applications require scaling mechanisms that exploit long-term deliberate planning to optimize performance and minimize cost.

3. *Heterogeneous resources are not fully exploited.* Most solutions scale a pool of same-sized virtual machines. Combining machines of different sizes and prices potentially lead to a better fit of the capacity to the demand. While previous approaches already manage heterogeneous resources, cloud applications can benefit from controllers that combine heterogeneous resources with long-term reasoning to schedule resources in a way that minimizes operational costs, according to cloud-defined billing periods.

4. *Most scaling controllers are oblivious to reconfiguration impacts.* Previous approaches address the migration of data after scaling actions. Performance degradation caused by data migrations is an important factor for adapting these systems. Yet, controllers rarely use models that estimate these impacts to make decisions about how or when to scale resources; instead, reconfiguration is often treated as a separate subsequent process.

5. *Scalability is a major bottleneck.* Controllers must explore large solution spaces of machine types and scaling actions in a long-term horizon to make optimal decisions. Applications required that elastic controllers are able to search those spaces efficiently, to respond to workload variations in a timely manner. How to engineer controllers that exploit deliberate planning and find solutions in real-time is an open question.

2.1.3 Deployment of Workflow Applications

Many multi-tier web and data processing jobs can be decomposed in a set of separate tasks, some of which can be executed in parallel, while others need to be executed in sequence. Each task can be executed in a single machine or may require the provisioning of multiple machines to be completed. The entire job is often subject to timeliness requirements, defined by the user. The most common way of representing this type of jobs is as a *workflow*. Workflow applications capture the desired execution order of the tasks in a job and can benefit from abstract reasoning to schedule the deployment of such tasks in dynamic environments.

Workflow applications can benefit from the flexibility offered by cloud providers. Indeed, workflow scheduling in the cloud has already been tested with homogeneous resources and predictable task completion times. In such scenarios, a controller can find a schedule that minimizes the makespan in order to meet the timeliness requirements of the application. Still, several opportunities offered by cloud providers remain somewhat unexplored.

Cloud providers offer a wide variety of virtual machines for diverse application types with pre-defined machine sizes. The combination of machine sizes and prices gives even more flexibility to execute workflows, assigning to each task the machines size that satisfies its computational requirements. Naturally, deploying tasks on different machines sizes would affect the execution times and the expected cost of the deployment. Thus, to schedule workflows in such scenarios, it is required to explore a larger search space of possible deployments, and minimizing the makespan may no be longer the main objective. In fact, several schedules that comply with the timeliness requirements may exist. The controller's objective would then be to select the schedule that incurs minimal costs to the application.

More interestingly, cloud providers offer revocable machines that can be rented at steep discounts from on-demand instances. Revocable instances may reduce workflow execution costs, at the penalty of added uncertainty in task completion times. This could be an impairment to the use of revocable machines when scheduling workflows with timeliness requirements. In particular, tasks deployed in revocable instances may fail if the instance is interrupted by the cloud provider before the task finishes its execution. In order to exploit revocable machines for the deployment on workflow applications, controllers must generate schedules that take into account the uncertainty in the outcome of a task execution. The added complexity could enlarge the search space and make more difficult the selection of the "best" schedule.

2.1.3.1 Related Work

The problem of workflow scheduling has deserved much attention by the research community. In the following, a summary of the most significant contributions to the state-of-the-art is presented.

2.1.3.1.1 Workflow Scheduling in the Grid The scheduling of workflows has been mostly studied in the context of *grid computing*, which concerns the use of widely distributed computer resources to reach a common goal and where each machine in the network is pre-assigned to the execution of a given task. The problem of workflow scheduling is, therefore, simplified, since the controller does not need to select different machines sizes, with pre-defined computational resources, for the execution of each task in a job. In the context of grid computing, some contributions deserve particular attention. In (Sakellariou, Zhao, Tsiakkouri, & Dikaiakos 2007), workflow applications with budget constraints are modelled as directed acyclic graphs (DAGs) and resolved using heuristics that, first produce good-performing schedules in terms of makespan or budget, and then combine these schedules to find a solution with a better trade-off between these metrics. In (Duan, Prodan, & Fahringer 2007), the problems of performance and cost optimization are formulated as sequential cooperative games and resolved using two algorithms: one, to minimize the expected execution time of workflow application; and two, to minimize the execution cost while guaranteeing a deadline. In (Yu, Buyya, & Tham 2005), the execution of workflow applications is scheduled using heuristics based on Markov Decision Processes (MDPs) and which objective is to minimize the execution costs while meeting time constraints. The common objective of these proposals is to design efficient heuristics to schedule workflows in order to satisfy goals in terms of execution time and execution costs.

2.1.3.1.2 Workflow Deployment in the Cloud Scheduling for the deployment of workflow applications in cloud environment has also deserved some attention. In (Kllapi, Sitaridi, Tsangaris, & Ioannidis 2011), a scheduling framework is proposed and tested with greedy, probabilistic, and exhaustive search algorithms. In (Mao & Humphrey 2011), scheduling of workflows with soft deadline constraints is achieved by heuristics that dynamically allocate and deallocate virtual machines and that execute tasks using the most cost-efficient instances. In (Byun, Kee, Kim, & Maeng 2011), the challenge of estimating the amount of resources required by each task

and the selection of virtual machines to deploy workflow applications is tackled with heuristic algorithms. Authors recognize that static allocation strategies lead to potential resource waste, since all scheduled resources may not always be required for the entire rental period. In (Zhang, Cao, Hwang, & Wu 2011), the authors tackle the issue of dynamic workflow scheduling on virtual clusters and reduce scheduling overhead with an iterative ordinal optimization approach that finds "good enough" schedules. In a similar line, (Maguluri, Srikant, & Ying 2012) addresses resource allocation problems for the deployment of real-time workflow applications using a stochastic models for load balancing and scheduling. These proposals exploit on-demand virtual machines in the cloud to minimize execution costs and focus on the design of heuristics for provisioning/allocating machines to the execution of tasks in workflows with deadlines.

2.1.3.1.3 Managing Revocable Instances Revocable machines have been offered by cloud providers for several years, yet heavy-tailed price distributions suggest that few applications actually use revocable instances. It is possible to conjecture that applications may be concerned by the possibility of too many job interruptions. In particular, to enable applications to use revocable instances it is important to answer two questions: how might the provider set the price, and what prices should applications bid? These questions are addressed in (Zheng, Joe-Wong, Tan, Chiang, & Wang 2015), where authors suggest that exploiting optimal bidding prices can be achieved by: (1) modeling the pricing scheme and matching the model to historically offered prices; and (2) deriving optimal bidding strategies for different job requirements and interruption overheads. The strategy of bidding optimal prices is, indeed, followed by most proposals that use revocable instances. However, some authors argue that sophisticated bidding strategies do not provide any advantages over simple strategies, since: (1) there are a wide range of bid prices that yield the optimal cost and availability; and (2) there is enough availability of resources in the public market. In fact, authors in (Sharma, Irwin, & Shenoy 2017) claim that cloud users can adopt trivial bidding strategies and focus instead on modifying applications to efficiently seek the lowest cost resources. This claim is supported by market models that demonstrate that, if the bid is equal to the on-demand price, then the expected cost is a fixed fraction β of the on-demand price (e.g. $\beta = 0.25$ for compute optimized instances in Amazon EC2) and the probability of being revoked is less than 80% in the first hour. Planning the execution of workflows applications using revocable machines in the cloud has not been fully studied (other related approaches are further discussed in Chapter 5).

2.1.3.2 Limitations and Open Issues

Research in the last decade tackled the problem of workflow scheduling to minimize the makespan, minimize execution costs, and meet timely requirements. However, there are still some limitations in current techniques to find optimal schedules as well as unexplored opportunities in exploiting the flexibility offered by cloud computing.

1. *Workflow scheduling algorithms are sub-optimal.* Most solutions focus on designing new algorithms and heuristics that can resolve the NP-hard scheduling problem in an efficient manner. While this is justified for real-time applications, not all workflows require online scheduling and, thus, are not bound by restrictive reaction times. Workflow applications with known structures could potentially benefit from optimal policies that use deliberate planning to search schedules that minimize the operational costs and meet deadline constraints. These policies could be defined offline, an be executed when deployment is required.

2. *Heterogeneous resources have not been fully exploited.* Previous techniques restricted by online scheduling times resort to resources that have been pre-selected or to a reduced number of machine types. Since resource selection is fundamental to the scheduling problem (e.g., a task executed in different machines have different execution times and costs), algorithms that do not consider all possible machine types are potentially unable to find the optimal solution.

3. *Revocable instances offer unexplored opportunities.* As suggested by previous market models, cloud applications could resort to trivial bidding strategies and still achieve optimal cost and availability. To take advantage of revocable instance prices, controllers must provide mechanism that support the dynamic provisioning of resources considering the possibility of task execution failures to occur due to instance interruptions. The resulting schedule must ensure that the workflow is successfully executed before the deadline and that execution costs are minimized. How to exploit revocable instances for the deployment of workflows in cloud environments is a challenging problem that has not yet been fully addressed.

2.2 Automated Planning for Self-Adaptation

2.2.1 Self-Adaptation and Autonomic Computing

The term *autonomic computing* was introduced for the first time in 2001. In this year, IBM released a manifesto (Horn 2001) identifying the increasing complexity in systems management as the primary obstacle for further development of the IT industry in the next years. Such consciousness led latter to IBM's vision of autonomic computing (Kephart & Chess 2003). Since then, the term is used to refer to any computerized system empowered to autonomously manage its resources. The goal of autonomic computing is to tackle the arising issues in management by integrating intelligence and autonomy into the operational process, while hiding intrinsic complexity to managers and staff.

The autonomic concept is inspired in biology. Specifically, it is inspired in the way the human autonomic nervous system unconsciously adapts the human body to its needs and to the environment, by taking care of intrinsic functions. Taking the human body as the reference, some characteristics were identified so that a system could be considered "autonomic" (Ganek & Corbi 2003), such as: the ability to know itself and its surroundings, to reconfigure under varying conditions, to optimize its performance, to recover from routine and malfunctioning, to guard from external attacks, etc. Such autonomic features, present in their biological namesakes, are known as *self-* properties* (Salehie & Tahvildari 2009).

At a primitive level, an autonomic system should exhibit *self-awareness*, the ability to be conscious of its states and behaviors, and *context-awareness*, the ability to reason about its operational environment and eventual changes on it. These rudimentary properties are essential for the system to interact with its environment and it is typically achieved by monitoring, i.e. collecting information. Once awareness is accomplished, the autonomic system should enable mechanism for supporting four major properties (Kephart & Chess 2003):

- Self-Configuration: to automatically and dynamically reconfigure system components and the relations among them, in response to contextual changes.
- Self-Optimization: to proactively seek for opportunities to improve its own performance and efficiency when external conditions change.
- Self-Healing: to spontaneously detect, diagnose and repair localized system failures.

From a more general perspective, a system exhibiting all such properties may be considered to be *self-adaptive*. The term *adaptation* is indeed used in biology to capture how organisms adapt to their environments, so as to maintain themselves in a viable state, through sensory feedback mechanisms. As such, an adaptive system may be defined as a set of interdependent and interacting entities, forming an integrated whole that together is able to respond to environmental changes or changes in the interacting parts. In the literature, however, many authors prefer to refer to such systems as *self-managing*; a term that is more specific to the intent to free system administrators from the details of system operation, i.e. to hide the operational complexity of the system.

Whereas self-adaptivity can be achieved in several ways, research supports the idea of enabling feedback loops as an external mechanism to control the operation of a system subject to environmental changes (Brun, Serugendo, Giese, Kienle, Litoiu, Müller, Pezzè, & Shaw 2009). The objective is to continuously adapt the system's output to follow or to converge to a specific desired behavior, usually given as an input reference to the system. Thus, to close the operational loop a system must be empowered with proper mechanisms: (1) to gather information and sense the environment; (2) to interpret the relevant data; (3) to identify the proper changes to the system required to adapt; (4) to decide what actions to take, with which intensity and in which order; and (5) to execute the action plan. Following this approach, numerous models and framework implementations have been proposed (Oreizy, Heimbigner, Johnson, Gorlick, Taylor, Wolf, Medvidovic, Rosenblum, & Quilici 1999; Dobson, Denazis, Fernández, Gaïti, Gelenbe, Massacci, Nixon, Saffre, Schmidt, & Zambonelli 2006; Garlan, Cheng, Huang, Schmerl, & Steenkiste 2004).

Arguably the most widespread feedback control model in the literature is the MAPE-K model (Kephart & Chess 2003). In this model, IBM differentiates the system (*managed system*) from its control mechanism (*managing system*), interfacing via: *sensors*, in charge of collecting data about the system, and *effectors*, to carry out adaptations to it. The model consists of four major (sequential) activities, depicted in Figure 2.1:

- Monitoring (M): responsible for collecting and aggregating data from sensors and converting them to behavioral patterns and symptoms.
- Analyzing (A): concerned with detecting the symptoms provided by the monitoring process, in order to recognize when the adaptation is required.

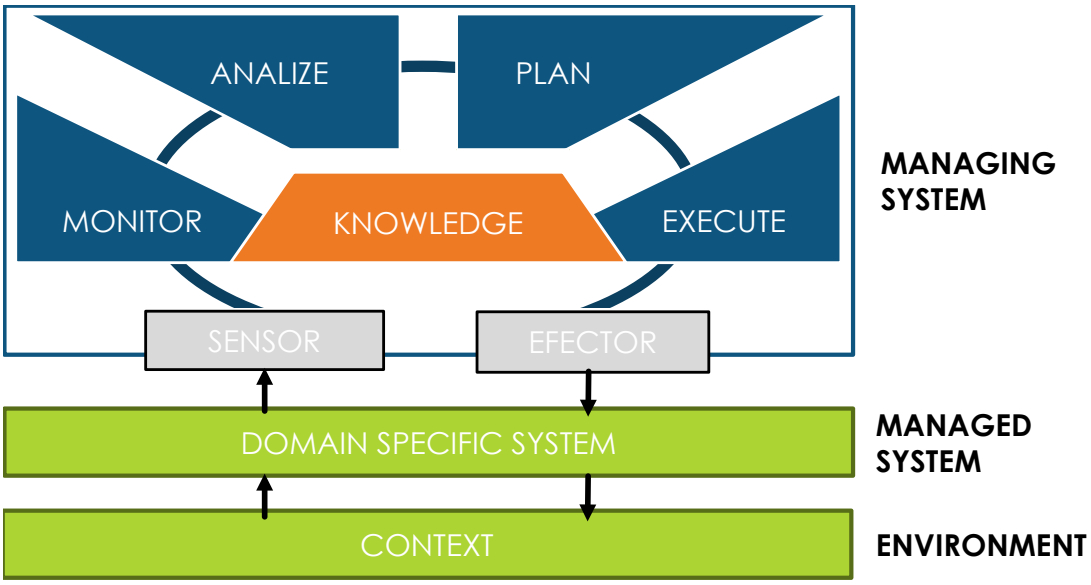

Figure 2.1: The MAPE-K Adaptation Framework

- Planning (P): determines how to adapt in order to achieve the best outcome, relying on certain criteria to compare adaptations and for the selection of a proper course of action.

- Executing (E): responsible for applying the actions determined by the planning process, by mapping such actions to what is provided by the effectors.

According to this model, a Knowledge (K) layer, which may contain system models, data, patterns, rules, commands, etc., serves as support for the four activities in the adaptation loop.

2.2.1.1 Architecture-Based Self-Adaptation

A common practice in the software engineering community is to approach the system from a conceptual model that defines its structure and behavior, usually in the form of a system architecture that comprises system components, the visible properties of those components and the relationships between them. In general, all architectural models tend to share the basic idea of the system being represented as a graph of nodes (components) and arcs (connections). Here, the components and connectors represent general concepts with some associated properties. For instance, in a client/server application, the components can be web servers, clients, and databases; connectors can be the links between them; and properties for a server can be the state “on” or “off”. Additionally, the model allows to set a number of architectural constraints

and binding properties on the components and connectors that define the valid configuration (and/or behavior) of the system.

Among several techniques to engineering self-adaptive systems, the architecture-based approach is recognized as being particularly well-suited for complex systems and has been widely explored (Dashofy, van der Hoek, & Taylor 2002; Chen, Peng, Yu, Nuseibeh, & Zhao 2014). The architectural model has been widely accepted as a mean to support reasoning about the system. The popularity of such approach can be explained because of its generality and level of abstraction, that makes its underlying concepts and the system description applicable to a wide range of domains. Besides, there exist a wealth of architecture description languages and notation that support dynamic architectures and formal architecture-based analysis and reasoning (Kramer & Magee 2007). In addition, many architecture-based adaptation frameworks have been proposed, including: Rainbow (Garlan, Cheng, Huang, Schmerl, & Steenkiste 2004), Plasma (Tajalli, Garcia, Edwards, & Medvidovic 2010), and Morph (Braberman, D'Ippolito, Kramer, Sykes, & Uchitel 2015), among others.

Another reason for adopting an architecture-based model is reuse. Reuse is sustained by the definition of *architectural styles* (Cheng, Garlan, Schmerl, Sousa, Spitznagel, & Steenkiste 2002), a set of principles that provide an abstract framework to describe the architecture for a family of applications (e.g. client/server applications). More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with the constraints on how to combine them. Additionally, architectural styles promote the reuse of style-specific adaptation mechanisms (e.g. actions, rules, plans, strategies).

2.2.1.1.1 Architectural Model From a broader perspective, the architectural models can support self-adaptation through the definition of:

- System Architecture: as component and connector with properties and constraints.
- System Metrics: as monitored performance indicators of the behavior of the system.
- Adaptation Actions: as operational instructions with effects on the architectural properties of the system and predicted impacts on the system metrics.
- Business Goals: that capture high-level business objectives about the behavior of the system, written in terms of architectural properties and system metrics.

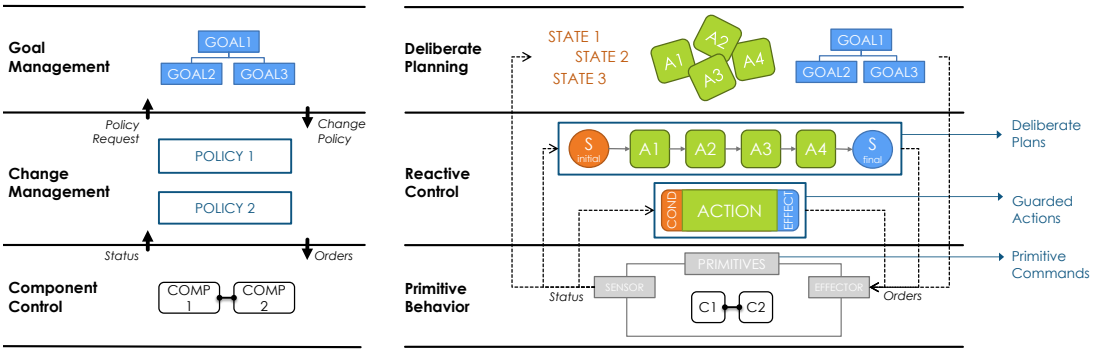

Figure 2.2: Layered Control and Operational Instructions

2.2.1.1.2 Layered Controls From the architectural perspective, an interesting approach to engineer self-adaptation consists in building up a hierarchy of control layers on the system; an idea inspired in the sense-plan-act architectures used in robotics (Gat & Bonnasso 1998). Three layers of control are proposed in (Kramer & Magee 2007):

- Component Control: feedback control loops that regulate the primitive behavior of the system (e.g. to adjust the operating parameters of the components).
- Change Management: reactive/corrective plan execution under arising conditions (e.g. to execute actions/plans that select new control behaviors and set new operating parameters).
- Goal Management: deliberate planning, time consuming computations that takes the current state and high-level objectives and attempts to produce plans to achieve those goals.

2.2.1.1.3 Operational Instructions Similarly, it is practical to understand operational instructions at different levels of control:

- Primitive commands: operational instructions that modify the system properties at the component-level (low-level of control).
- Guarded actions: operational instructions in the form of condition-action-effect constructs, used to correct the system behavior (middle-level of control).
- Deliberate plans: operational instructions in the form of a sequences of guarded actions, used to guide the system to a desired state defined by business goals (high-level of control).

The Stitch language (Cheng & Garlan 2012) refers to these categories as *operators*, *tactics*, and *strategies*, correspondingly.

2.2.1.2 Adaptation Policies

Adaptation policies describe how to adapt the system under common conditions. They notably transform high-level business objectives that define the correct behavior of the system into actual operational instructions that steer such behavior under common conditions. Adaptation policies have been extensively utilized to engineer adaptive systems (Ghezzi, Pinto, Spoletini, & Tamburrelli 2013).

Adaptation policies are best described in three parts: conditions, operations, and effects. Adaptation conditions trigger the execution of the operations and refer to both *system conditions*, i.e. the configuration of the system in terms or architectural properties, and *environmental conditions*, i.e., the external factors that affect the performance of the system, notably captured by the system metrics. Adaptation operations refer to the operational instructions in form of an *action* or a *plan* that shall be executed under such conditions. Adaptation effects are the expected observable changes in behavior of the system once these operations are enacted and refer to both the expected *effects* in the configuration of the system and its architectural properties (e.g. activating a server S1 changes a property S1.state from "off" to "on") and the predicted *impact* on the performance of the system, captured by the system metrics (e.g. activating a server S1 improves the latency by 20%).

The use of policies for the purposes of self-adaptation is widely accepted by the community. Adaptation policies are attractive due to their: (1) support to reasoning, since they capture understanding of the system from a conceptual perspective; (2) readability to humans, which can be crucial for human involvement in system operations (e.g. to accredit accountability for the system's behavior); and (3) efficiency, as they guarantee timely executions at run-time operations. However, policies can be hard to maintain, especially in business contexts subject to frequent changes, such as, the introduction of new component types, the emergence of new adaptation conditions and operations, or even the variation of the business preferences regarding the system behavior.

2.2.2 Automated Planning

Planning, as a fundamental property of intelligent behavior, is an object of research by the artificial intelligence community. Automated planning is a branch of this discipline that concerns the realization of strategies or action sequences, typically for execution by intelligent agents.

2.2.2.1 Planning Problem

The planning problem varies according to some factors regarding the system under study.

- Deterministic/non-deterministic/stochastic: A system is considered deterministic if its transition function is a deterministic function of the input action and the previous state (i.e, given the same action and same previous state, the system goes to the same next state). It is considered non-deterministic if, given an action and a previous state, the next state is not uniquely determined. The system is considered stochastic if it is non-deterministic and the probability of transition to the various next states is known.
- Static vs. dynamic: A system is said to be static if its transition function becomes the "identity function" when the control input is said to "No op". More informally, the world doesn't change when the agent is not doing anything. A system is said to be dynamic if it is not static. One can, of course, talk about a spectrum between static and dynamic.
- Observable vs. non-observable: A dynamic system is said to be "completely observable" if its internal state can be recovered through the output transformation function. If only part of its state can be observed, then it is called "partially observable". The control theoretic definition of observability actually considers a system to be observable if the state of the system at time t can be completely recovered through observations on the state of the system at a finite number of future time instances.

2.2.2.1.1 Problem Definition Self-adaptive systems are considered to be dynamic systems empowered with monitoring tools that make them fully or partially observable. The effects of the actions over the system can be modeled as deterministic, non-deterministic, or stochastic, on a case to case basis, depending on the complex system interactions, the environmental changes and the generalized uncertainty.

The automated planning community has standardized the definition of planning problem as deterministic (*classical planning*) or stochastic (*probabilistic planning*). Additionally, the effects of actions may take in consideration temporal factors, in which case, the problem is considered to be a *temporal planning* problem. In the following, the planning problem is defined from an architectural perspective of the system.

1. *Classical Planning:* Classical planning considers the deterministic transformation of the domain. Given a set of states S, a set of actions $A : S \rightarrow S$, a subset of actions $A(s) \subseteq A$ applicable in each state s, an initial state $s_0 \in S$, and a set of goal states $S_G \subseteq S$, classical planning is the task of finding a sequence of actions that, when applied to s_0, yields one goal state. In this definition: S is the set of all valid system states (e.g. the configuration of the system in terms of architectural properties and the metrics the indicate the performance of the system); A is the set of all possible actions, with associated conditions and (deterministic) effects, in terms of the architectural properties of the system and the metrics that indicate the performance of the system; s_0 is a state of the system that does not satisfy the declared architectural constraints and/or high-level business goals; and S_G is the set of all system states that satisfy the declared goals.

2. *Probabilistic Planning:* Probabilistic planning considers the probabilistic transformation of the domain for stochastic systems. Given a set of states S, a set of actions A, a transition probability function $P : S \times A \rightarrow S$, an initial state $s_0 \in S$, a set of goal states $S_G \subseteq S$, probabilistic planning is the task of finding a sequence of actions that, when applied to s_0, yield one goal state with certain probability. In this definition: S is the set of all valid system states; A is the set of all possible actions, with associated conditions in terms of the architectural properties of the system and the metrics that indicate the performance of the system; P is the transition probability function specifying the probability $P(s, a, s')$ of reaching state $s' \in S$, when action $a \in A$ is applied to state $s \in S$; s_0 is a state of the system that does not satisfy the declared architectural constraints and/or high-level business goals; and S_G is the set of all system states that satisfy the declared goals.

3. *Temporal Planning:* The problem could consider the time when actions are executed, usually via a sequence of decision steps $l = 1, 2, ..., L_{max}$, where L_{max} is referred to as horizon. Temporal planning deals with time as a continuous exogenous variable and is particularly useful for problems characterized by actions with variable duration and concurrency.

2.2.2.1.2 Planning under Uncertainty Most systems are naturally affected by uncertainty. Uncertainty can manifest itself in the form of unexpected events or changes in the system and its environment, such that predicting the state of the system after the execution of an action becomes a difficult task. The malfunctioning of one of the components of the system may prevent an action of having its expected effect on one of the system properties (e.g. if a server S1 fails to boot up, switching on that server may end up in a property S1.state equal to "off"). Also, an unexpected change in the environment may affect the performance of the system in a undesired way (e.g. a spike in the load of a newly activated server may saturate its processing capacity). The non-deterministic nature of these changes acts in way that the accumulated effects of executing a (planned) sequence of actions, can deviate the system's behavior to an undesired final state. Planning under uncertainty is indeed an open issue and subject of study in the automated planning community.

In order to provide these system with control mechanisms, it is a common practice to model the system in a stochastic way by defining a planning problem with probabilistic transitions. For instance, a malfunction of the system may lead to an undesired effect with 1% probability, or environmental changes may lead the system to a peculiar variation in its performance metrics with 5% probability. The definition of a probabilistic problem ensures that decision-making takes uncertainty in consideration when finding and selecting a plan that can lead the system to a desired state with a certain probability. To control stochastic systems, it is important that decision-making mechanisms are capable of keeping track of the changes that follow the execution of each action, such as to steer the coarse of action in the right direction whenever the system deviates from its expected common behavior.

2.2.2.1.3 Planning with Utility and Preferences System operations are typically associated with a sense of benefit or cost to the system's performance (and the business, in general). The execution of an operational instruction may improve the performance of the system at the expense of monetary costs (e.g. switching on a new server may add computational capacity to a service and also increase the incurred costs), and otherwise. Also, an operation may potentially lead the system to a (transitional) state that is particularly beneficial or harmful to the system. Naturally, when selecting a coarse of action that leads the system to a goal state, the controllers would select the plan that does not violate system constraints and that result in more rewards and/or less costs to the system, overall.

In order to capture these operational preferences, in addition to the definition of goals, a reward function R (or a cost function C, depending on the case) is often associated to the sates of the system $s \in S$ and the actions $a \in A$ (and transitions) that transform such states. Therefore, when there are several desired goal states and various sequences of actions that may lead to one of them, these rewards serve as an indicator of preference to support the selection of the "best" (or cheapest) sequence of actions that leads to a goal state. Here, the term "best" is typically defined in terms of a utility that must be optimizes. In addition, hard preferences can be imposed as constraints that regulate which actions and state are transited during the selection of a plan.

2.2.2.1.4 Complexity of the Planning Problem The definition of the planning problem, ideally, captures all the complexity associated to making deliberate decisions that can guide the system to satisfy the business preferences and goals. The complexity of the problem increases with the number of possible states of the system (i.e. the combination of all configuration parameters and system metrics), the number of actions that can be executed to each state, the number of transitions between states and the number of time steps in the horizon. This increase is not linear, but exponential (combinatorial), and leads to the *state space explosion* problem, where the size of the state space is so large that even containing it represents a computational challenge and search time become impractical.

The definition of a planning problem that can accurately model the complexity of decision-making and that is also practical in terms of search times is, therefore, an engineering problem.

2.2.2.2 Planning Techniques

2.2.2.2.1 Planning with Artificial Intelligence (AI) The artificial intelligence field is rich in models, languages, and tools for planning. The Planning Domain Definition Language (PDDL) is the de-facto language of automated planning (McDermott, Ghallab, Howe, Knoblock, Ram, Veloso, Weld, & Wilkins 1998). At its simplest, it specifies the planning problem via object *types* and their boolean properties in the form of *predicates*, their *initial state* and *goals*, and the *actions* that could change the state of the world. Extensions of the language have been made to support: optimization of a plan-*metric*, temporal properties via declaration of *durative actions* (PDDL2.1), and planning with *objects fluents* i.e. continuous values consumed or produced by

actions (PDDL3.1). Defining a planning problem in PDDL is done in two parts: a *domain specification*, where object types, predicates, functions, and actions are declared; and a *problem specification*, where objects are instantiated and initialized and goals are declared. In terms of tools, the International Planning Competition (IPC) (Vallati, Chrpa, & McCluskey) serves as a platform to compare the effectiveness and efficiency of search algorithms and heuristics. In this platform, planners are tested over a set of example problems described in a standardized language and planners are ranked according to their effectiveness and efficiency. The competition is composed of several tracks including deterministic, probabilistic, and temporal.

2.2.2.2.2 Markov Decision Processes Markov Decision Processes (MDPs) describe scenarios where the system "moves" through a sequence of states. The transition between states happens in response to a sequence of actions taken by an agent, at different decision steps. The outcome of actions are typically non deterministic, such that an action can lead the system to several system states with different probabilities. Executing an action on the system brings some reward (or cost, depending on the case). The objective is to control the system by taking appropriate action while optimizing some criterion (e.g. maximizing the total expected reward over a sequence of decision steps). The most significant property shared by systems described as MDPs is the *markovian* property, meaning that any action applied to a state in a given decision step is independent from the past, i.e. no information about previous decisions may influence a current decision. From an algorithm perspective, fundamental algorithms such as *value iteration* and *policy iteration* are commonly used to solve MDPs in their most generic form: the Stochastic Shortest Path (SSP) problem. There exist, though, a multitude of approaches proposed by AI researchers to scale solution algorithms to larger problems. For a deeper understanding of MDPs, the reader is referred to the literature in (Mausam & Kolobov 2012).

2.2.2.2.3 Linear Programming Linear programming is an approach typically related to operational research. Models in linear programming consist of a symbolic representation of the objects that compose the system under study, connected according to certain rules. It manipulates the symbolic representation of the building blocks of the system until a satisfactory results is obtained. Formally, linear programming is a technique for the optimization of a linear objective function, subject to linear equality and inequality constraints (Dantzig 2016). Linear programming often deals with the deterministic transformation of the world, in discrete

amounts. Certain probabilistic problems can also be reduced to linear programming problems, e.g., scheduling problem (Baptiste, Le Pape, & Nuijten 2012). It has proven useful in modeling diverse types of problems in planning, scheduling, assignment, and design. Additionally, a multitude of constraint solvers currently exists that are efficient at finding solutions to large problems (e.g. CPLEX (IBM 2016)). Linear programming benefits from efficient exploration of large solution spaces, in detriment of more abstract and flexible representations (e.g. like those captured by AI planning languages).

2.2.3 Automated Planning for Self-Adaptation

As discussed above, automated planning is rich in: (1) definitions, to classify planning problems according to the factors involved; (2) languages, to represent the behavior of the system, the environment, and the control actions; and (3) tools, that implement heuristics and exhaustive search algorithms to explore the space of possible solutions and to select the best plan. Therefore, systems that require adaptation may resort to automated planning techniques to support decision-making. In fact, for a system to be truly self-adaptive, it must be able to decide which actions are more appropriate to execute under certain conditions and in which order to execute them to reach a state that satisfies the business-defined goals; the task known as Planning in the MAPE-K adaptation framework (Kephart & Chess 2003).

2.2.3.0.1 Making vs. Achieving Two main approaches can be identified for introducing planning capabilities into software systems (Horn 2001). The first one is to engineer self-adaptivity into the system at the design phase. The second is to achieve self-adaptivity through adaptive learning. In (Sterritt 2003) these two approaches are called *making* and *achieving*:

- Making: Has an implied system and/or software engineering view to engineering adaptivity into the system. In general, the system triggers the adaptation process as the result of a undesired conditions (e.g. a constraint violation) and executes actions according to pre-defined rules, to correct the operation of the system.
- Achieving: Has an implied artificial intelligence and adaptive learning view to achieve adaptive behavior. By achieving, instead, the system enable intelligent planning mechanisms on-the-fly to select an appropriate course of action that guides the system from the current state towards a desired state.

These two approaches do not necessarily contradict each other. Indeed, it is possible to see both as cooperating at different levels of control; making, at a reactive control level (e.g. change management in (Kramer & Magee 2007)) and achieving, at a deliberate control level (e.g. goal management in (Kramer & Magee 2007)). Whereas the former permits quick reaction to expected changes, the latter performs a slower deliberate planning to achieve system goals. A system able to adapt at both level, reactive and deliberative, is desired.

2.2.3.0.2 Online vs. Offline In general, since deliberate planning is a time consuming task, it is not always possible to plan on-the-fly. Indeed, planning can be done online or offline:

- Online Planning: is recommended for systems with lower complexity and loose time requirements, such that an adaptation plan can be found in short time without compromising the performance of the system. Online planning generates a plan that is specific to the initial state of the system when the planning task is activated.
- Offline Planning: is recommended for highly complex systems (e.g. when planning must consider many possible combinations of objects, actions, and transitions) that require a fast response. Offline planning can generate adaptation plans that are triggered reactively during online operations.

2.2.3.0.3 Reactive vs. Proactive The dynamics of the environment can also affect the type of planning mechanism to put in place. Depending on the adaptation scenario, planning can be reactive or proactive.

- Reactive Planning: is applicable in scenarios where the environment changes slowly and steadily, such that the system has enough time to react to the instantaneous events that triggered the planning task (e.g. the violation of a constraint).
- Proactive Planning: is able to predict the behavior of the system and the environment in a future horizon and make decisions before these changes take place. Thus, this technique is better suited for scenarios where environmental changes are fast and predictable and where there is limited time to plan and execute, such that corrective measures must be taken in advance.

2.2.3.1 Related Work

2.2.3.1.1 Adaptation Policies Adaptation policies can be derived manually, using knowledge from experts, administrators and managers exclusively. For instance, Stitch (Cheng & Garlan 2012) is a language that is able to capture operational instructions dictated by system experts, in the form of guarded actions and deliberate plans. However, expert-defined plans may be limited in the number of covered system conditions or in the way actions are combined together to reach a desired behavior.

Alternatively, reactive policies can be derived automatically, as shown by previous work. In (Rosa, Rodrigues, Lopes, Hiltunen, & Schlichting 2013), a general framework for automatically deciding adaptation actions is presented. Rules are generated offline, specifying component adaptations that may help achieve high-level business goals, and selected online, evaluating the conditions of the system. Similarly, Fossa (Frömmgen, Rehner, Lehn, & Buchmann 2015) uses a methodology that separates the adaptation logic from the actual application implementation, to learn event-condition-actions rules by automatically executing a multitude of tests. Rule sets are generated by algorithms such as genetic programming and the results are evaluated using a utility function provided by the developer. These solutions do not exploit the benefit of deliberate planning, though.

Solutions that generate adaptation plans at run-time also exist. For instance, a system which permits arbitrary dynamic adaptation by exploiting reactive plans is presented in (Sykes, Heaven, Magee, & Kramer 2007). Reactive plans are generated online with a planning tool from high-level goals given by the user, and the behaviour of the system is described by the set of condition-action-rules given in the plan. Plans generate at run-time cannot be validated by experts held accountable for the system and may be applicable only to the initial system state that triggered adaptation, instead of common system conditions.

2.2.3.1.2 Planning with AI The idea of modeling architecture-based self-adaptation as a planning problem is not new. Previous attempts to encode software architectures into PDDL include: a methodology to specify architectural reconfiguration as actions in PDDL via graph transformations (Tichy & Klöpper 2012; Rasche & Ziegert 2013); a framework for automated generation of processes for self-adaptive software systems based on workflows, AI planning and model transformation (da Silva & de Lemos 2011); and a support method to encode ADL

types an constraints in PDDL representation (Méhus, Batista, & Buisson 2012). Previous work also demonstrates that AI languages and tools can be used effectively in specific scenarios. For instance, automated planning can be used to support the automatic generation of paths for the evolution of software architectures. The evolution problem is mapped into a planning language capable of representing the initial and target architectures, the evolution operators, the path constraints and the functions that evaluate the qualities of the path; and, then, off-the-shelf planners derive the candidate evolution paths. This approach is proposed in (Barnes, Pandey, & Garlan 2013), where authors use the PDDL language and two planners (LPG-td (Gerevini, Saetti, & Serina 2006) and OPTIC (Benton, Coles, & Coles 2012)) to resolve a data migration problem under business constraints. This work puts in evidence the difficulty of writing correct specifications in PDDL, the limited features of planning tools, and the long search times. Additionally, the authors do not address how to plan under uncertainty or how to assemble more complex strategies that combine candidate paths.

2.2.3.1.3 Planning under Uncertainty In the literature, it has been suggested that stochastic search techniques might be effective to engineer planning for self-* systems (Coker, Garlan, & Le Goues 2015). The design of self-* systems inherently involves trade-offs in multiple interrelated and evolving dimensions, including complex notions of time, cost, and non-determinism, that stochastic techniques could help resolve. The authors present a prototype to the generation of plans offline, based on genetic programming combined with a probabilistic model checker used as a fitness function. Proactive self-adaptation under uncertainty has been tackled in (Moreno, Cámara, Garlan, & Schmerl 2015). An auto-regressive time series predictor is used to estimated the future behavior of the environment. Then, a simplistic model based on a three-point discrete-distribution approximation is used to construct a short-term prediction of the environment and its uncertainty. In (Cámara, Garlan, Schmerl, & Pandey 2015), the authors present an approach to automatically synthesize optimal reactive plans via model checking of Stochastic Multi-Player Games (SMGs), that enables modeling uncertainty both as probabilistic outcomes of adaptation actions and through explicit modeling of environmental behavior. In a similar line, (Franco, Correia, Barbosa, Rela, Schmerl, & Garlan 2016) proposes to improve self-adaptation planning through software architecture-based stochastic modeling. In particular, it proposes a formal automated approach to translate a specification from an Architecture Description Language (ADL) to a Discrete Time Markov Chain (DTMC), as well as a method to predict

the quality impact of each adaptation strategy that supports decision-making in unexpected or untested conditions. A hybrid planning approach that combines deterministic planning with Markov Decision Process (MDP) planning is proposed in (Pandey, Moreno, Cámara, & Garlan 2016). Deterministic planning provides plans quickly when timeliness is critical, while MDP planning allows the system to generate optimal plans when there is sufficient time to do so.

2.2.3.2 Limitations and Open Issues

1. *Reactive policies do not benefit from deliberate planning.* Automatically generated policies are limited to event-condition-action rules that can be executed greedily. Adaptation plans are generated and executed online, instead. This gap can be narrowed with the automated generation of reactive policies in the form of deliberate plans, applicable under common system conditions.

2. *AI planning languages and tools are limited.* Reasoning about complex decision-making for systems self-adaptation requires high level of expressiveness, which is offered by planning languages. However, planning tools are limited in the number of language features they support. Additionally, AI planning tools require excessive time to find a solution to large scale problems. This could prevent the use of these techniques online. The question of how to use AI automated planning to support online adaptation remains unanswered.

3. *Planning under uncertainty is time consuming.* Effective adaptation demands the exploration of large solution spaces, defined by the combination of system objects, adaptation actions, and state transitions expected in the temporal horizon. Long-term planning under uncertainty is complex, computationally expensive and time consuming. Currently, there are no generic planning languages and tools that can model this type of complexity and be resolved efficiently at run-time.

2.3 Evaluation Metrics

To evaluate the design of controllers that exploit automated planning to decide the deployment and management of applications in the cloud it is important to define a set of evaluation metrics:

1. *System Requirements:* Computer systems often have requirements in terms of their configuration. These requirements define the valid configurations and parameters of the system, that must not violated by the application manager. At times, they can be expressed in the form of *dependability constraints*. For instance, an interactive application in the cloud may impose that at least one server always remains active during reconfiguration, to guarantee some availability to the users. A correct controller is one that does not violate the requirements of the system.

2. *System Performance:* Systems often monitor their operations via key performance indicators. Interactive server/client applications serving content to users, typically have two main performance indicators: throughput and latency. *Throughput* is the actual rate that information is transferred. *Latency*, or response time, is the the delay between the user request and the user response, which is affected by the network bandwidth and the processing capacity of the servers. Applications may have as a business objective to maintain the throughput above a certain threshold or to minimize the response time. Workflow applications may have other performance metrics, such as the *execution time*, the time required for the execution of all tasks in a job. For instance, these application can express the business goals by defining bounds to the execution times (i.e. deadlines). A correct controller is the one that does not violate the performance goals and/or that leads the system to the state that optimizes the performance according to the user preference.

3. *Operational Costs:* A major reason why software applications resort to dynamic provisioning of resources in the cloud, is the reduction of operational costs. While operational costs is a rather broad concept, in the context of cloud computing services it is possible to concentrate in the monetary costs (in dollars) billed to the application by the cloud providers; specifically, the monetary costs due to consumed resources. In addition, applications often resort to the concept of "utility" and the monetized penalties due to violations of the system requirements. A controller must minimize the monetary costs or maximize the

utility. It is worth noting that controllers must often balance between performance and costs, according to user preferences.

4. *Controller's Responsiveness:* Given that applications in the cloud are exposed to dynamic environments, controllers must have the quality of reacting quickly and positively to environmental conditions. For reactive scaling controllers, responsiveness consists in deciding and executing an action immediately after an unexpected event has occurred. If the controller is supported by policies, these must have reaction plans to cover the arising conditions; failing to come up with a solution in limited time is a violation of the responsiveness property. For proactive scaling controllers, responsiveness consists in being able to act before the (undesired) changes take place, as well as being able to correct the decided actions if unexpected changes arise. Finally, for workflow schedulers, responsiveness consists in being able to come up with a schedule that ensures that execution times are respected, as well as being able to adapt these schedules if unexpected conditions arise.

5. *Controller's Scalability:* For a controller to find the "best" solution, it must handle the complexity of decision-making problem. A controller is scalable if, given a description of the factors that affect decision-making, it is able to generate the entire space of possible solutions, explore it efficiently, and find the "best" solution. In particular, online controllers must be able to search the space and find a solution in limited time. Controllers that are unable to consider all possibilities and fail to search the space for complex problems are not scalable.

3 Reactive Policies for Elastic Scaling

This chapter introduces and evaluates an approach to automatically generate reactive policies for elastic scaling of cloud applications. In particular, it addresses the question of how can *classical planning* be used for the (offline) generation of policies that support elastic scaling in cloud environments.

This chapter discusses how to integrate automated planning into the generation of high-level adaptation policies that: (1) can be easily derived from the system models and human expertise; (2) are effective at guiding the system towards the best reachable configuration; and (3) can react to common conditions. Automated planning tools have been used as a complement to human decision-making, as they have proved effective at exploring the combinatorial space of actions to find plans. Additionally, novel automated mechanisms to build comprehensive policies that cover interesting conditions and to facilitate keeping policies up-to-date have been incorporated.

3.1 Motivation and Goals

Currently, most solutions to elastic scaling consider a homogeneous resources and are controlled by restrictive reactive policies that do not make use of deliberate decision-making. The most significant evidence of this phenomenon is the way cloud providers allow applications to scale their resources. For instance, Amazon Auto Scaling [1] allows applications to define reactive policies that regulate the server pool via scaling actions that are triggered by events (e.g. when the average CPU utilization reaches a pre-defined threshold). Ideally, controllers for elastic scaling can take advantage of the flexibility offered by cloud providers in terms of heterogeneous resources. A pool of servers that combines different virtual machine sizes may fit more tightly the requirements for processing capacity. Table 3.1 gives an example of a set of general purpose instances, with distinct processing capacity, prices and start-up times.

[1] https://aws.amazon.com/autoscaling/

Table 3.1: Cloud Resources: Amazon EC2 Instances - m1

Instance Size	Short	CPU	Price ($/hr)	Start-up (sec)
m1.small	S	1 cores	0.044	98
m1.large	L	4 cores	0.175	95
m1.xlarge	XL	8 cores	0.350	100

The Importance of Planning: Controllers for elastic scaling can also exploit the benefits of deliberate planning. To discuss the benefits of planning, Fig. 3.1 illustrates a scenario where the initial state of a system is that of one active *m1.xlarge* server, a configuration that is highly expensive but that serves users in low response times. Let's assume this states is reached after a small decrease in the workload occurs. Then, scaling down the system could provide savings. To make the best decision, operators must deal with some challenges:

A *Selecting the target configuration:* Given heterogeneous resources, multiple configurations may provide the capacity required to deal with a change in the observed load. For a simple scenario depicted before, Fig. 3.1 shows three final configurations with a slightly different trade-off between the expenditures and the (predicted) response time observed by clients after the system is reconfigured. As several valid final configurations are possible, the decision on the "best" one is dictated by business preferences regarding performance and expenditures.

B *Selecting the best adaptation path:* The order in which adaptation actions are executed to guide the system towards the target configuration is very important. Experts may want to prevent system downtime during reconfiguration ensuring that at least another server is up before the *m1.xlarge* server is decommissioned; this would force the system to temporarily further increase expenditures before a final (cheaper) state is reached. Fig. 3.1 depicts three different paths that can be followed to reach the third final state; these paths illustrate how different orderings could produce very different transient states during adaptation.

C *Taking time into account:* It is worth noting that adaptation paths to different final configurations may take different time to execute. In some settings, the time required to execute a reconfiguration must also be taken into account when adapting the system to ensure the fastest plan can be found. Since classical planning does not account for actions with duration, temporal planning may be better fitted for problems with this complexity.

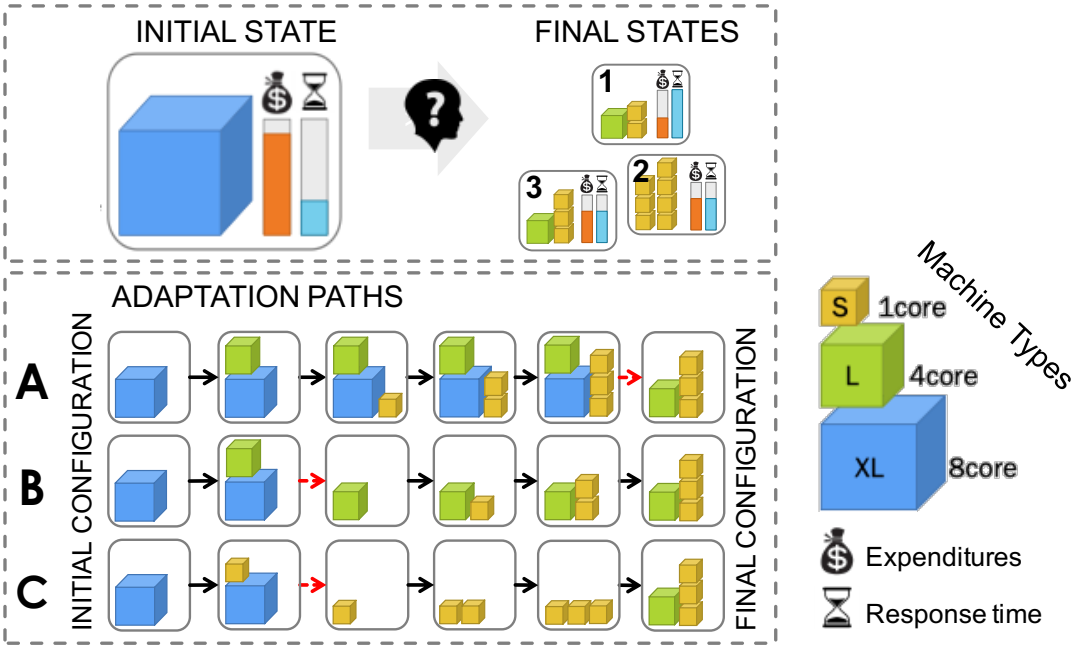

Figure 3.1: Motivation: Planning target configuration (1, 2 and 3) and best adaptation path (A, B and C)

AI planning can do a great deal to resolve decision-making in such scenarios, by providing means to evaluate target configurations according to business preferences, to select the best adaptation path in compliance with system constraints, and to provide tools supporting temporal actions to ensure that the fastest plan is selected.

Still, AI planning tools exhibit intrinsic limitations that must be circumvented. First, planners are designed to find plans from a specific initial state. Instead, when generating policies, a comprehensive set of adaptation rules should cover all undesirable conditions that the system can plausibly reach. Second, planning tools are designed to deal with boolean properties. Therefore, previous proposal are unable to resolve the challenges imposed by the temporal nature and numeric complexity of the scenario described before (see Related Work).

Goal: To use classical (and temporal) planning for the (offline) generation of policies that support elastic scaling of interactive applications running in cloud environments. In particular: (1) to combine a description of the problem (easily derived from human experts) with the use of automated tools for the search of complex strategies; (2) to exploit deliberate planning to find solutions that are effective at guiding the system towards the best reachable configuration; and (3) to build techniques to encapsulate plans as adaptation policies that cover common system conditions and that facilitate the revision of such policies in changing business contexts.

3.2 Approach

This contribution addresses the unresolved challenges of exploiting automated planning for the generation of high-level policies in adaptation scenarios that rely heavily on temporal and numeric properties. From the perspective of architecture-based self-adaptation, automated planning can rely on the definition of an model that describes the system operations and the environment. In the following, it is assumed a system model that defines: the *system architecture* as component and connector types, with their associated properties and constraints; *metrics*, as system performance indicators; *actions*, as guarded operations with effects on architectural properties and impacts on metrics; and *goals* over architectural properties and metrics.

High-level policies are generated by combining three complementary tasks, namely: i) an *encoding* task that translates the system architecture, adaptation knowledge, and the initial system state into a language specification readable to the planner; ii) a *planning and selection* task, that evokes the planner multiple times to obtain a set of suitable solutions and selects the preferred one according to some optimization criteria; and iii) a *scanning* task, that selects the system states to which planning is applied and merges plans under common system conditions. The interplay of these tasks is illustrated in Fig. 3.2.

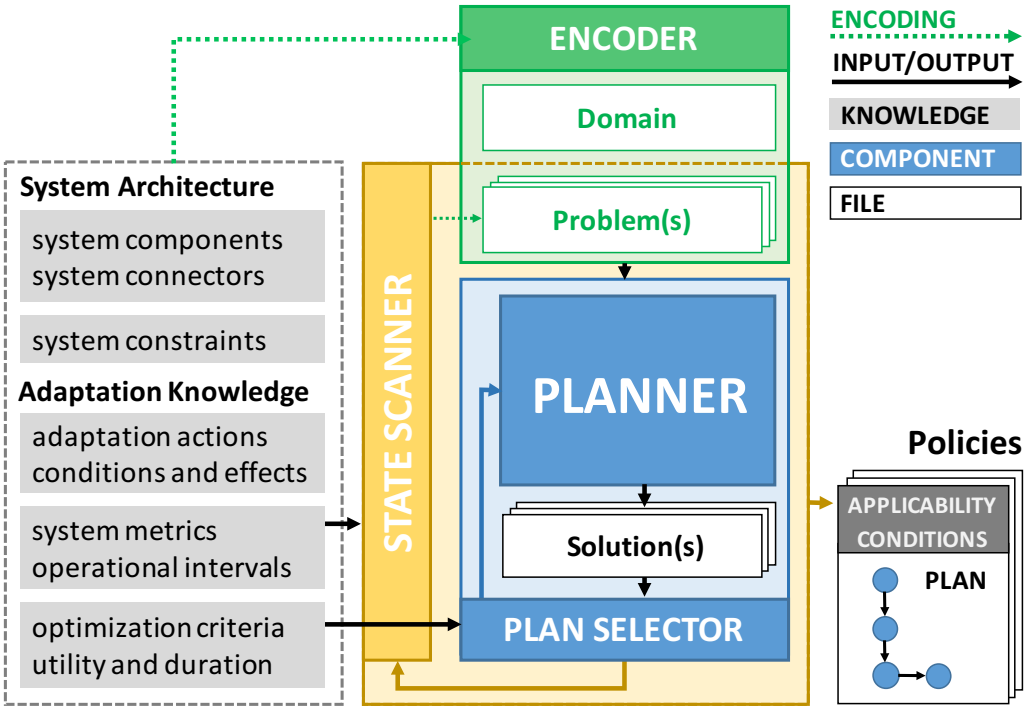

Figure 3.2: Approach Overview

To illustrate how this approach can generate reactive policies, consider the adaptation scenario presented in Fig. 3.1. The approach is applicable to cases in which new system conditions or configurations are available; in our scenario, consider that there is a new dependability constraint that enforces a minimum of four processing cores at all time. Initially, the space of system configurations and system metrics to be considered for adaptation is defined by the expert and taken as input to the scanner; in our scenario, the system configuration is defined by the possible combinations of three server types (*m1.small*,*m1.large* and *m1.xlarge*) and the system metric of interest is the *response time*. The optimization criteria capturing the business goals is assumed to be the minimization of a utility function defined by the expert; in our scenario, the business goal is to minimize a combination of *expenditures* and *response time*. Then, the scanning task is used to smartly select initial system states such as to avoid exhaustive exploration of the entire space; e.g. it considers an initial configuration of one active *m1.xlarge* server when the *response time* is low, between 0ms and 20ms. Each state is encoded into a problem specification and the planner and selector are used to obtain adaptation plans; e.g. the planner generates two adaptation plans that meet the new dependability constraint, A and C (showed in Fig. 3.1), and the selector chooses plan C to be best at minimizing the utility. This process is iterative, as the space is scanned deeper in certain regions, selecting new initial states depending on the outcome of previously visited states; e.g. if the initial *response time* is very low, under 10ms, another adaptation plan D (similar to C, but that does not add a third *m1.small* server) leads to a better utility. The scanning task finishes by generating policies, such that adaptation plans are executed given some applicability conditions on the system metrics; in our scenario, for a configuration of one active *m1.large* server, two policies are generated: (1) if *response time* $\leq$ 10ms, then execute plan D, and (2) if 10ms $\leq$ *response time* $\leq$ 20ms, then execute plan C.

3.2.1 Encoding

In order for planners to be able to find solutions they must be fed with an encoding of the system model and adaptation problem in a standard action language. It is worth mentioning that this approach targets adaptation scenarios that rely heavily on numeric properties to guide adaptation. Mapping the system architecture (components, connectors, and their properties) into PDDL (McDermott, Ghallab, Howe, Knoblock, Ram, Veloso, Weld, & Wilkins 1998; Fox & Long 2003) is straightforward. A mapping into PDDL is shown in Fig. 3.3.

Architectural Model Concepts	PDDL Language Constructs	Example Specification
Component/connector: types	Types (and sub-types)	(:type server)
Component/connector: properties (boolean)	Predicates: type-specific boolean functions	(:predicate on ?s - server)
Component/connector: properties (numeric)	Functions: type-specific numeric fluents	(:functions cpu ?s - server price ?s – server)
Component/connector: instances	Objects: typed instances declared statically	(:object S0 S1 S2 - server)
Metrics: component-specific (boolean)	Predicates: type-specific boolean functions	(:predicate overloaded ?s - server)
Metrics: component-specific (numeric)	Functions: type-specific numeric fluents	(:functions load ?s - server)
Metrics: system general (numeric)	Functions: numeric fluents	(:functions expenses)
Metrics: thresholds (numeric)	Functions: numeric fluents	(:functions budget)
Action: declaration and duration	Durative Action: action name Parameters: type-instances to be affected Duration: action-specific numeric value	(:durative-action TurnOff-Server :parameters (?s - server) :duration (= ?duration 100)
Action: conditions (components and metrics)	Condition: list of boolean/numeric conditions	:condition (and (on ?s) (< (load ?s)(* 0.2 (cpu ?s))))
Action: effects (components and metrics)	Effect: list of predicate/function assignments	:effect (and (not (on ?s)) (decrease expenses (price ?s)))
Goals (components and metrics)	Goals: list of boolean/numeric conditions	(:goal (< expenses budget))

Figure 3.3: Approach: Encoding in the PDDL planning language

Nonetheless, it is not obvious how to encode system metrics (numeric functions) or the optimization criteria, such that they can be used to guide the search for the best adaptation path. Action languages were originally crafted to deal with boolean properties, not numeric functions. Consequently, the primitive versions of the language allow only to accumulate numerical effects (impacts) on a unique cost function, instead of separate numeric functions (metrics). Also, goals need to be expressed as a conjunction of predicates, instead of conditional expressions on numeric functions (metrics). This approach exploits more advanced versions of the language (PDDL 3.1 (Gerevini & Long 2005)) to overcome these restrictions.

In particular, this approach requires language support for:

- *Numeric and object variables* (a.k.a. fluents) enable the definition of system metrics (numeric functions) that can accumulate numerical effects (action impacts) over a path.
- *Arithmetic operators* enable the definition of simple action impact functions over system metrics and can be used to declare more elaborate system metrics (e.g. a linear combination of other metrics)
- *Conditional expressions* permits to evaluate conditions over fluents (e.g. if a system metric is over a threshold), which is particularly useful for evaluating action preconditions, conditional effects, or goals over system metrics.
- *Universal and existential preconditions* serve to extend the expressiveness of conditional expressions, allowing to test if a condition applies to “at least one” or to “all” components of a certain type in the system.

- *Conditional effects* extend action impacts so as to declare collateral effects.

The optimization criteria (duration and utility) is then encoded into the language. Plan duration optimization is inherent to temporal planning, enabled by the definition of *durative actions* and search heuristics to find the fastest solution. Plan utility optimization is enabled by the language by declaring a metric minimization statement, that optimizes a *total-cost* function over the path. Such utility function can be defined as a linear combination of the normalized metrics that are to be minimized, an instantiation of the weighted sum method to multi-objective optimization (Marler & Arora 2010). However, while the metric minimization feature is supported by classical and temporal planners in general, no current planner that supports other requirements of the model (e.g. numeric variables) supports such feature as well.

3.2.2 Planning

The planning task relies on off-the-shelf tools as a main building block. Temporal Fast Downward (TFD) (Eyerich, Mattmüller, & Röger 2009; Vallati, Chrpa, Grzes, McCluskey, Roberts, & Sanner 2015) is a progression search-based system that extends classical planning to support durative actions and numeric variables. TFD is particularly outstanding due to support to numeric variables, which makes it an adequate solver for the scenario presented in the motivation.

TFD, however, is limited when dealing with plan-metric optimization, as the planner forces plan duration as the unique optimization criteria. Therefore, search heuristics driven by duration criteria (relying on the TFD planner) are combined with plan optimization based on utility criteria, via the *plan selector*.

Particularly, for a given initial state, the plan selector generates problem specifications declaring different utility goals of the form *(:goal (≤ (utility) (threshold))*. The utility function is defined as a linear combination of the normalized metrics to be minimized. A binary search is used to select the minimum possible threshold in a utility range, initially defined between zero and the utility computed at the initial state. In each round of the binary search, the planner is executed to find a plan that leads to a final state whose utility is less than the selected threshold. The search terminates when the selected threshold is sufficiently low, such that the obtained plan can be considered good enough, and/or when utility goals for lower thresholds are found unattainable.

3.2.3 Scanning

The generation of reactive policies assumes that high-level plans that cover general conditions of the system must be found offline. This ensures that deliberate planning can be incorporated without harming the timely response of the system. The general conditions covered by a given plan are a subset of relevant states that the system may fall into at run-time.

To derive offline plans for general conditions of the system, the multi-dimensional space of system states that are relevant for adaptation must be searched (scanned). A system state consists of a configuration (specifying which components are active, how they are connected, and their properties) and a particular combination of the system metrics. Since the space state is in general too large, it is impossible to run the planner for all possible states. Scanning consists of exploring the metric space given an initial configuration of the system, as well as exploring the space of all possible initial configurations.

Scanning Metrics Space: This mechanisms explores the metric space for a given system configuration c. The metric space is characterized by having D dimensions, one dimension per metric. Each metric takes values in a numeric continuous interval $[d_{min}, d_{max}]$ provided to the algorithm by the expert. The algorithm takes two additional parameters as inputs: x, the number of initial divisions of each metric interval, and μ, the target granularity of the space.

It is worth mentioning that this approach assumes that the n-dimensional metric space is mapped into a utility space (via an expert-defined function) for decision-making, so it comes natural to resort to utility to predict space granularity. For example, consider two system states, $s1$ and $s2$, with utilities $u1$ and $u2$ respectively. When $|u1 - u2|$ is smaller than a certain utility threshold μ, such states are indistinguishable in the utility space, thus, a finer division would be undetectable and purposeless for decision-making. Parameters x and μ enable user control over the initial segmentation of the space and the depth of the scanning, and can be tuned to trade-off between accuracy of the results and efficiency of the algorithm. The scanning algorithm for the metric space (Algorithm 1 presented bellow) ensures that the multi-dimensional space of all system metrics is uniformly visited and that scanning is focused into interesting regions, avoiding exhaustive exploration.

Algorithm 1 Scanning Metrics

input: a configuration c, ordered set of dimensions D, ranges $[d_{min}, d_{max}]$ for each dimension, number of points per dimension x, granularity for the utility μ
output: a set of plans P for all visited system states
function SCANNING(c, D, x, μ)
  for all $d \in D$ **do**
    for all $i \in [0, x-1]$ **do**
      $t[i] \leftarrow i * (d_{max} - d_{min})/x$
    end for
    $m[ordinal(d)] \leftarrow \{t[i] : i \in [0, x-1]\}$ ▷ define x points
    $r[ordinal(d)] \leftarrow \{(t[i], t[i+1]) : i \in [0, x-2]\}$ ▷ define ranges
  end for
  $V \leftarrow \{(m_1, m_2, \ldots m_{|D|}) : m_i \in m[i]\}$ ▷ define vertexes
  $H \leftarrow \{(r_1, r_2, \ldots r_{|D|}) : r_i \in r[i]\}$ ▷ define partitions
  $S \leftarrow \{(c, v) : v \in V\}$ ▷ define initial states
  $P \leftarrow \emptyset$ ▷ define plan set
  $P \leftarrow$ PLANNING(S, P) ▷ find plans for all initial states
  while $H \neq \emptyset$ **do**
    $h \leftarrow e : e \in H$ ▷ select partition h
    $H \leftarrow H \setminus \{h\}$
    $E_h \leftarrow edges(h)$ ▷ get edges in partition h
    for all $e \in E_h$ **do**
      $(v1, v2) \leftarrow vertexes(e)$ ▷ retrieve edge vertexes
      if $|utility(c, v1) - utiliy(c, v2)| > \mu$ **then** ▷ check granularity
        $p1 \leftarrow P(c, v1)$
        $p2 \leftarrow P(c, v2)$ ▷ retrieve plans of edge vertexes
        if $p1 \neq p2$ **then** ▷ compare plans of edge vertexes
          $bisect \leftarrow true$ ▷ decide to bisect partition h
          $break$
        end if
      end if
    end for
    if $bisect$ **then**
      for all $d \in D$ **do**
        $(t1, t2) \leftarrow get(h, ordinal(d))$ ▷ retrieve points of d in h
        $tm \leftarrow (t1 + t2)/2$ ▷ compute midpoint in h
        $m_h[ordinal(d)] \leftarrow \{t1, tm, t2\}$ ▷ define points in h
        $r_h[ordinal(d)] \leftarrow \{(t1, tm), (tm, t2)\}$ ▷ define ranges in h
      end for
      $V_h \leftarrow \{(m_1, m_2, ...m_{|D|}) : m_i \in m_h[i]\}$ ▷ define vertexes in h
      $H_h \leftarrow \{(r_1, r_2, ...r_{|D|}) : r_i \in r_h[i]\}$ ▷ define partitions in h
      $S_h \leftarrow \{(c, v_h) : v_h \in V_h \setminus V\}$ ▷ define unvisited states in h
      $P \leftarrow$ PLANNING(S_h, P) ▷ find plans for unvisited states in h
      $S \leftarrow S \cup S_h$ ▷ update states
      $V \leftarrow V \cup V_h$ ▷ update vertexes
      $H \leftarrow H \cup H_h$ ▷ update partitions
    end if
  end while
  return P
end function

Algorithm 1 consists of two steps:

The **first step** proceeds to the segmentation of the metric space uniformly. For each dimension, the algorithm selects x uniformly distributed points within the operational interval of each metric, thus dividing it into $x-1$ smaller ranges equal in size (line 5-10). A combination of ranges in D defines a *partition* h, i.e. a n-dimensional hyper-cube representing a sub-set of possible metric values within the space. Each partition is confined by its vertexes and edges; each *vertex* v defined as a unique operational point of the system, i.e. a combination of specific values for all metrics. Thus, the segmentation results in a first set of x^n vertexes V and a set of $(x-1)^n$ partitions H of the entire space (line 11-12). Then, a set of initial system states S is found; each *state* s defined as a pair of the initial configuration c and an operational point v (line 13). The algorithm executes the *planning* routine for such first set of states S and get in return a plan p to each initial state s (line 14-15). As a result of this first step, a set of plans P is generated for $(x)^n$ operational points evenly spread in the metric space.

The **second step** proceeds to the iterative division of the metric space when needed. This step checks that all vertexes in a given partition h have the same plan p and, if not, proceed to separate such partition into smaller ones; this is done iteratively until the granularity of the partitions is "small enough". The algorithm proceeds by selecting a partition h from H (line 17-18). Then, for all edges in h, it compares the plans generated for adjacent vertexes $p1$ and $p2$ (line 19-30). It decides to bisect the space if these plans are different for any pair of adjacent vertexes in h (line 25-28). Bisection is done by computing the midpoint for every range defining the current partition h (line 32-37). Then, it generates all new (sub-)partitions and their corresponding vertexes in h, the sets H_h and V_h (line 38-39). It defines new states s, as the combination of c with the newly generated vertexes v_h (not yet visited) and execute the *planning* routine to find plans for all such new states (line 40-41). It updates the sets S, V, and H (line 42-44). This procedure is repeatedly executed for all partitions in H and stops when rounds cease generating new partitions (i.e. no more bisections are done). This occurs when partitions get 'small enough' or when adjacent plans are the same.

Algorithm 1 returns a set of reconfiguration plans $P(c)$ for each sub-region of the metric space, for configuration c (line 47).

Scanning Configuration Space: Initially, the expert defines the available system component types with their properties and the architectural constraints stating how such components are put together (e.g. the maximum number of available instances of the same type or the maximum budget). Defining all system configurations, i.e. possible combinations of active components and property settings, is either too demanding or unnecessary. Yet, operators tend to have some insights about which are the most common system configurations, a knowledge worth availing.

It is possible to scan the configuration space taking as a seed a small set of common system configurations defined by the expert C_0. This approach takes advantage of the metric space scanning algorithm (described previously) to predict successive system configurations that can result from the adaptation process. This is done using the set of reconfiguration plans $P(c_0)$ returned by the algorithm. For any plan $p(s_0) = [a1, a2, ..., af]$ in $P(c_0)$ for a initial state $s_0 = (c_0, v)$, the algorithm is able to generate a set of subsequent system states $\{s1, s2, ..., sf\}$, from which it derives a set of subsequent system configurations $\{c1, c2, ..., cf\}$. In the first iteration, the procedure generates subsequent system configurations for all seed configurations c_0 in C_0. In next iterations, the procedure takes as input the result of a previous iteration C_i to generate a new (bigger) set C_{i+1} of subsequent states worth exploring. The procedure is repeated for a number of iterations k (a parameter set by the expert), allowing the user to control the adaptation horizon in the number of subsequent possible configurations to visit.

Consolidating Policies: To generate policies, the solution consolidates adaptation plans under common system conditions. The algorithm merges adjacent partitions (with shared hyperplanes) that share common adaptation plans. It extract system conditions r_m as the per-metric ranges under which a plan is valid. This results in a set of policies, defined by: initial configuration c, system conditions r_m, and adaptation plan p.

3.3 Case Study

To study the effectiveness of this approach, policies are generated to support elastic scaling of a web content application, a news website similar to CNN. This user-interactive application is served by a pool of machines in the cloud that it must adjust to meet the capacity requirements of the user demand, which varies over time.

Let's consider an adaptation scenario where the cloud provider make available three different machine types, as shown in Table 3.1: *m1.small*, *m1.large*, and *m1.xlarge*. The application expert has defined two new dependability constraints: (1) at least one server has to be active at all time to guarantee service availability, and (2) the system processing capacity cannot be lower than six cores at any time.

For illustration purposes, in the following the system is considered to depart from an initial configuration in which two *m1.xlarge* servers are active and the performance indicator of interest is the response time, when in the range between 0ms and 50ms.

3.4 Application

3.4.1 Encoding

System components: The available server instance sizes (small, large, and xlarge) are specified with their associated prices, start-up times, cpu capacities, and shutdown times as shown in Listing 3.1. A server instance can be enabled or disabled, which is captured as a boolean property. The system expenditures, i.e. the cost of all enabled servers, and the system capacity, computed as the sum of the capacity of all enabled servers, are encoded as functions. While the system architecture consists of more components and connectors in a real cloud application, Listing 3.1 shows only the server types for simplicity.

System metrics: The model considers the response time, i.e. the average latency perceived by clients, as the metric of interest, which is encoded as a function (line 5 in Listing 3.1).

System goals: The utility function is defined as a linear combination of the system expenditures (configuration-specific) and the response time (metric-specific). The weights and normalization factors are declared to each term (line 6 in Listing 3.1). Expert-defined weights are: $w_1 = 0.14$ for expenditures and $w_2 = 0.86$ for response time. Normalization factors are taken as the maximum value of the operational interval defined by the expert: expenditures in [0.000$/hr, 0.800$/hr] and response time in [0ms, 4000ms]. The planning goal to find a system state which utility is below the utility threshold is declared (line 7 in Listing 3.1).

Listing 3.1: PDDL specification: components metrics and goals

```
1 (:types        small large xlarge - server)
2 (:predicates   (is_enabled ?s - server))
3 (:functions    (price ?s - server) (cpu ?s - server) (capacity)
4                (startup ?s - server) (shutdown) (expend))
5 (:functions    (time))
6 (:functions    (w1) (w2) (n1) (n2) (th))
7 (:goal (<= (+ (*(/(expend) n1) w1) (*(/(time) n2) w2)) (th)))
```

Listing 3.2: PDDL specification: action DeactivateServer

```
1  (:durative-action DeactivateServer
2   :parameters  (?s - server)
3   :duration    (= ?duration (shutdown))
4   :condition   (and
5                (at start (is_enabled ?s)))
6                (over all (exists (?s - server) (is_enabled ?s)))
7                (at start (>= (- (capacity) (cpu ?s)) (cputh))))
8   : effect     (and
9                (at start (not (is_enabled ?s)))
10               (at end   (decrease (expend) (price ?s)))
11               (at end   (assign (time)
12               (* (/ (capacity) (- (capacity) (cpu ?s)) (time))
13               (at end   (decrease (capacity) (cpu ?s))))
```

Systems actions: Adaptation actions are two: activate or deactivate server (see Listing 3.2). Each action is defined by its: *parameters* as the specific component type to be affected; *duration* which is parametrized to be equal to the start-up/shutdown time; *conditions* for the action to be selected (e.g. an instance must be active in order to be deactivated - line 5); *effects* on the configuration parameters; and *impacts* expressed as a linear function (e.g. the impact on the response time is inversely proportional to the variation in the capacity - line 11).

Dependability constraints: Since constraints are not supported explicitly by the planning language and tools, they are encoded as action conditions (see Listing 3.2). The dependability constraint imposes that at least one server remains active during plan execution as a condition to deactivating servers (line 6). Also, a server cannot be deactivated when the predicted final capacity (after shutdown) is less than a threshold cputh = 6 cores (line 7).

3.4.2 Planning

The encoded utility function to guide plan selection assigns the value $(0.14) * (expend/0.800) + (0.86) * (time/4000)$ to a state with a configuration that costs *expend* per hour and has the value *time* for the response time (a value known to be in [0ms, 50ms]). Several utility thresholds are generated using the procedure described before to find an adaptation plan. Experiments show that the solution, in average, is found between the 5th and 6th iteration.

3.4.3 Scanning

As mentioned before, for the case study the initial configuration c consists of two enabled XL servers (and, hence, expenditures of 0.700$/hr). The scanning procedure is run for the one-dimensional space for response time in the interval [0ms, 50ms]. The algorithm is run considering: the initial number of points is set to $x = 11$ and the utility granularity $\mu = 0.001$. In the first step, the metric interval is partitioned in 10 sections of $5ms$. In the subsequent phases, deep scanning is done for neighboring states with mismatching plans. The algorithm further partitions the range [20ms, 25ms] in subsequent iterations, thus visiting two new states. In total, the metric space scanning algorithm visits 13 initial states for configuration c. Plans are merged under common conditions, resulting in two policies; one plan covers system response time in range [0ms, 24ms] and a different plan covers the range [24ms, 50ms].

3.5 Evaluation

3.5.1 Generated Policies

The generated policy for the initial system configuration c and metric conditions r_m in the sub-range [0ms, 24ms] is shown in Listing 3.3. The adaptation plan aims at a final configuration consisting of one large server (L1) and three small servers (S1, S2, S3), for a total of seven CPU cores to handle the workload.

The adaptation actions are ordered to meet the dependability constraints in terms of minimum processing capacity and number of active servers. As a consequence, XL1 server cannot be decommissioned right after XL2; this ensures at least one server is active at all time. Similarly,

XL1 server is deactivated only after L1, S1, and S2 are active; so the plan ensures a minimum compound capacity of six cores at any time. It is worth noting that a similar set of actions scheduled in a different order could violate dependability constraints. Similarly, alternative valid plans (e.g. activating several small server instead of the one large server) could be costlier or slower in execution.

Listing 3.3: One of the Generated Policies

```
CONFIGURATION: (XL1 enabled) (XL2 enabled)

conditions: 0ms < response time < 24ms
adaptation plan:
(A) DeactivateServer  (XL2)
(B) ActivateServer    (L1)
(C) ActivateServer    (S1)
(D) ActivateServer    (S2)
(E) DeactivateServer  (XL1)
(F) ActivateServer    (S3)
```

3.5.2 Scalability

The TFD planner took 4.54 seconds in average to find the solutions and 12.4 seconds to generate the plans presented above, for a problem instantiating 17 objects. In the evaluation, a problem specifications with an increasing number of server instances is considered. The results suggest that search times in TDF increase dramatically with the number of objects (see Table 3.2). Long search times could limit these tools to find solutions at run-time.

Table 3.2: Scalability: Search Times of TDF Planning Tool

N. Instances	Instance types	Search Time
26	18 S + 5 L + 3 XL	4.62 sec
31	19 S + 8 L + 4 XL	7.38 sec
33	20 S + 10 L + 5 XL	46.1 sec
36	23 S + 10 L + 5 XL	114 sec

Time performance is correlated to the characteristics of the search graphs (specific to the encoded problem), search algorithms, and search heuristics implemented by the planning tools. TFD heuristics benefit the exploration of the action space over that of object space. Thus, by declaring actions that are parametrized with properties (e.g. prices, start-up times), the encoding already stressed TFD's search capabilities. Yet, the encoding could be tailored to

reduce TFD's search times, e.g. by using typed-based properties (instead of object-based) and non-parametrized actions. Since this approach is supposed to be executed offline, any planner that is able to find adequate solutions, even if it takes several minutes or a couple of hours, is still useful for this approach. TFD, in particular, was able to find solutions in all tested scenarios.

3.5.3 Discussion

The approach ensures that the system requirements, e.g. dependability constraints, are encoded in the specification of the planning problem, such that the generated policies do not violate these requirements. The approach also encodes a multi-objective goal that balances the system performance (e.g. reduced response time experienced by the customer) and the operational costs (e.g. expenditures due to rented machines), according to user preferences.

An elastic controller empowered by the generated policies would be responsive to expected conditions of the system. Policies are generated offline and can be executed timely, in scenarios where the variation in the workload are slow. Also, the generated policies ensure that expected conditions are covered, since the scanning task already explores the space of all possible configurations and environmental conditions, defined by the experts.

The planning tools could resolve the complexity of real-life scenarios. Yet, AI planning tools still present limitation that prevent them to be used online. Search times increase significantly with the number of objects to explore. Nevertheless, the applicability of the approach is not affected by long search times, since the policy generation procedure is executed offline.

3.6 Related Work

Most decision-making mechanisms for elasticity are based on greedy solutions that select first the action with best immediate impact on the system's performance. Most commercial cloud providers, e.g. Amazon, Google, Microsoft, and several academic proposals, e.g. (Rosa, Rodrigues, Lopes, Hiltunen, & Schlichting 2013; Cheng & Garlan 2012), resort to these solutions. These greedy approaches can fail when faced with complex scenarios that require planning.

Few planning-based approaches exist currently. For instance, Mistral (Jung, Hiltunen, Joshi, Schlichting, & Pu 2010) relies on a holistic search algorithm, a variation of A*, that considers adaptation costs to determine optimal plans. Still, adaptation actions do not have associated impact functions on performance metrics and, thus, derived plans cannot account for transient violations of dependability constraints during adaptation.

Previous solutions have used AI planning for self-adaptation, similar to the presented approach. For instance, in (Barnes, Pandey, & Garlan 2013) the authors encode a software architecture evolution problem in the PDDL language and resolve it with temporal planners. In PLASMA (Tajalli, Garcia, Edwards, & Medvidovic 2010), architectural descriptions are automatically translated into PDDL and a planner is used to find the components that need to be included in a configuration to achieve the system functional goals and, then, to find a plan able to transform the current configuration to the desired one. Also, an approach for the generation of temporal plans is proposed in (Ziegert & Wehrheim 2015) that allows durative reconfigurations and proposes an encoding into PDDL that solves concurrency issues. However, these solutions lack the ability to tackle problems that require numeric values (e.g. system metrics) to guide the search. Additionally, the generated plans are not aggregated in the form of adaptation policies that are applicable under common system conditions.

3.7 Conclusions

The approach presented above aims at responding the question: How can classical planning be used to support the (offline) generation of policies for elastic scaling in cloud environments?.

The proposed approach solves a classical planning problem (extended with temporal features) using AI planning. This approach was successful at meeting the goals and addressing

limitation of previous work in that:

1. The generation of policies is supported by automated tools that can foresee unexpected combinations of machine types and scaling actions to elaborate (more complex) strategies. It departs from an encoding of the problem easily derived from the system model and human expertise. It improves on previous work that relied solely on humans to define reactive policies.

2. The generated policies exploit the benefit of planners to choose a sequence of actions to guide the system to the desired configuration. In this sense, it improves on previous work where elastic scaling was reduced to a greedy execution of event-condition-rules, potentially unable to lead the system to its "best" configuration.

3. The encoding of the planning problem is flexible enough that new machine types and system constraints can be easily incorporated and the planning and scanning steps can support the generation of a revised set of policies. That way, the proposed solution can assist the frequent revision of policies due to changes in the context.

Summary

This chapter has presented the implementation of an automated planning solution to support the definition and revision of reactive policies for elastic scaling in the cloud, and its evaluation via simulations in a case study. This solution relies on AI planning languages and tools to generate policies that support elastic scaling. It is effective at mapping a (simplified) architectural model of the system into a standard action language (PDDL3.0), at scanning the space of system conditions, at generating a set of plans produced by off-the-shelve planners (TFD), and at selecting the best configurations and plans according to multi-objective criteria. The results suggest that automated planning is a valid alternative for the offline generation of policies applicable to real life cloud environment scenarios. Notably, planning can circumvent the limitations of human-defined policies, producing more elaborate combinations of actions that lead the system to the desired configuration, while preventing the violation of system constraints.

Publications

The work presented in this chapter contributed to the following publication:

Automated Generation of Policies to Support Elastic Scaling in Cloud Environments. R. Gil Martinez, A. Lopes, L. Rodrigues Proceedings of the ACM Symposium on Applied Computing (SAC). Marrakech, Morocco. 2017

4 Proactive Scaling and Reconfigurations

This chapter introduces and evaluates an approach to execute automated planning at run-time for elastic scaling and reconfiguration of cloud applications. In particular, it addresses the question on how can *temporal planning* be used for the proactive reconfiguration (online) of interactive applications in cloud environments.

This chapter explores the reconfiguration of cloud-enabled applications using controllers that combine proactive techniques with the ability to manage heterogeneous resources. The solution relies on temporal patterns obtained from historical data to predict the evolution of the workload and to initiate adaptation before the service quality is affected. Decisions are made based on knowledge about the workload curve, the cloud resources, and the initial system configuration. The chapter presents Augure, a controller that uses constraint solvers at run-time to search the space of actions in long-term horizons and to select a plan that: (1) minimizes the price billed by the cloud provider and (2) mitigates the negative side-effects of reconfiguration on the service quality. In addition, to study proactive controllers that make (greedy) decisions looking at short-term horizons, this contribution introduces Vadara+, an extension of Vadara (Loff & Garcia 2014) that manages heterogeneous resources.

4.1 Motivation and Goals

Applications in the cloud may be exposed to environments where changes occur rather abruptly and the time for planning on-the-fly is limited. Reactive policies that execute greedy guarded actions may respond quickly; yet, without a perspective of future events, the system may be exposed to oscillations that hinder its performance. Reactive policies that rely on deliberate planning, presented in the previous chapter, are also oblivious to the future variations of the environment and may divert the system's behavior in an undesired way. In scenarios as such, proactive planning can be an appropriate solution.

The contributions presented in this chapter are motivated by three key features: (1) proactive planning as a solution to narrow reaction times; (2) the importance of acknowledging reconfiguration impacts; and (3) the ability to manage heterogeneous resources; discussed below.

Proactive Scaling Proactive controllers have advantages when compared to reactive ones. First, they can scale resources in a timely manner. Reactive adaptation is slowed down by long monitoring and activation times (including server boot-up, configuration, and state migrations). For instance, Netflix waits 5-10 minutes before triggering activation and servers can take up to 45 minutes to be ready to serve requests (Jacobson, Yuan, & Joshi 2013). Proactive controllers can perform early activations; thus, servers can be ready before the (expected) surges occur. Second, proactive scaling can anticipate state migrations. While reactive activation may induce migration to occur when the system is highly stressed, proactive techniques can execute migrations when the system is not stressed. Fig. 4.1 shows throughput degradation due to delayed reactions and how early activations can mitigate these impacts, when scaling resources of an elastic database system for online transaction processing applications (see Testbeds: E-Store). Finally, proactive controllers also benefit from longer times to explore larger solution spaces.

Naturally, the efficiency of proactive scaling depends on having accurate predictions of future workload. Luckily, many user-interactive applications see predictable workload behaviors. Wikipedia reports daily access with decreased volumes at night (Urdaneta, Pierre, & van Steen 2009). Netfilx sees high demand after releasing new episodes (Jacobson, Yuan, & Joshi 2013). Sports websites see surges during important events (Arlitt & Jin 2000). The New York Stock Exchange transactions rise an order of magnitude during the first and last ten minutes of a trading day (Nazaruk & Rauchman 2013). Travel agencies, flight operators, and e-commerce stores see seasonal boosts during summer and holidays. In scenarios like these, proactive adaptation is convenient (Qu, Calheiros, & Buyya 2018; Kim, Wang, Qi, & Humphrey 2016).

Previous approaches to proactive scaling use temporal predictions of the workload curve to make short-term scaling decisions, in a greedy manner (see Related Work). Differently, Augure benefits from long-term decisions on the predicted workload curve to derive scheduled plans. Time series analysis has also been used in the past to identify behavioral patterns in the workload (Herbst, Huber, Kounev, & Amrehn 2014; Liu, Liu, Shang, Chen, Cheng, & Chen 2017). Augure sits on the assumption that workload patterns exist and can be learned.

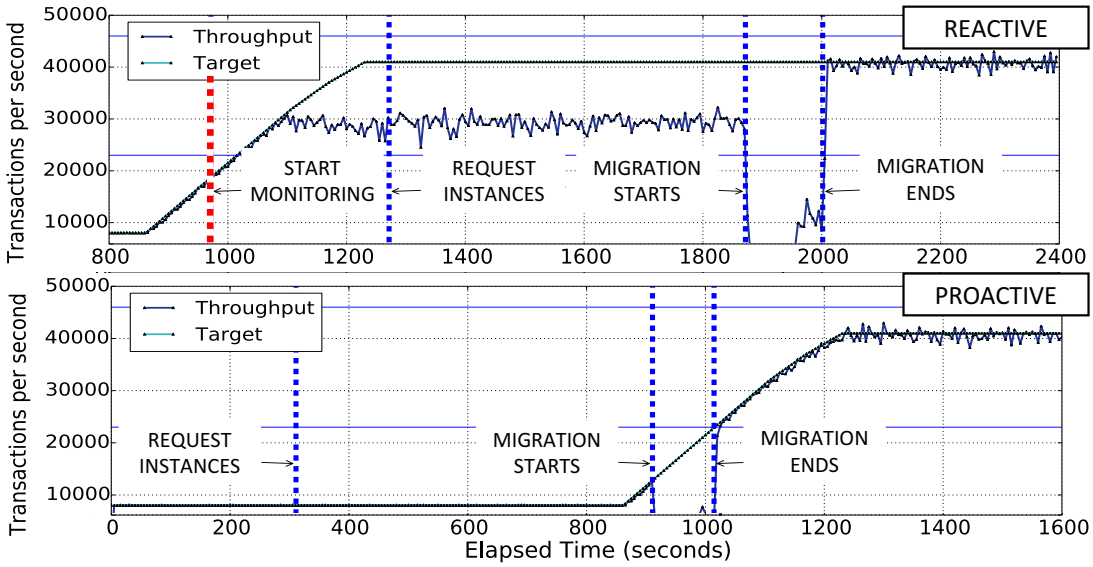

Figure 4.1: Motivation: Proactive scaling and reconfiguration of stateful applications. Experimental testing with E-Store

Reconfiguration Impacts Scaling actions can penalize the system performance while being executed. This is relevant in stateful applications, where scaling actions may require data to be transferred among replicas. The state transfer not only consumes network resources, but also processing capacity in active replicas, contributing to increased latency and reduced throughput (Taft, Mansour, Serafini, Duggan, Elmore, Aboulnaga, Pavlo, & Stonebraker 2014). Despite this, a large majority of scaling solutions for the cloud are oblivious to these effects. In previous solutions, like Vadara (Loff & Garcia 2014), reconfiguration impact can be taken into account by starting actions ahead of time, thus ensuring that migrations end before the system is overloaded. By making the controller aware of reconfiguration costs, it is possible to find plans that mitigate service degradation when operating in heterogeneous settings. A plan with mixed server types can reach resource utilization at minimal cost (equivalent to that reached using equal-capacity servers of the smallest type); yet, it benefits from using less servers and triggering less adaptations, thus, reducing their impact on service quality.

Heterogeneous Resources Heterogeneous resources offer an opportunity to mix a set of machines of distinct sizes that can tightly adjust to the required capacity. Previous approaches that manage heterogeneous resources do not take into account temporal factors to find a plans to better fit the workload curve in a long-term horizon. Since pricing schemes by cloud providers often charge by the hour, having a long term view may help decide differently about a composition of the pool of heterogeneous resources that could reduce costs more efficiently.

Goal: To use temporal planning for the (online) generation of proactive plans for elastic scaling and live reconfiguration of applications in cloud environments. In particular: (1) to enable proactive planning techniques that can anticipate environmental changes and find plans to scale resources accordingly; (2) to employ long-term decision-making, to make scaling decisions able to reduce monetary costs by accounting for billing schemes that charge resources in long periods (e.g. by the hour); and (3) to enforce awareness about the reconfiguration impacts, to mitigate the negative side-effects on the system performance.

4.2 Approach

This section introduces Augure, a proactive controller for live reconfiguration of cloud services that leverages knowledge about workload patterns and takes into account the heterogeneity of resources and the impact of reconfiguration actions when planning adaptation. Augure works by combining four complementary tasks: (1) pattern extraction, (2) fitting and prediction, (3) planning and, (4) execution, as shown in Fig. 4.2. The notion of workload patterns and each of the tasks performed by Augure are described below.

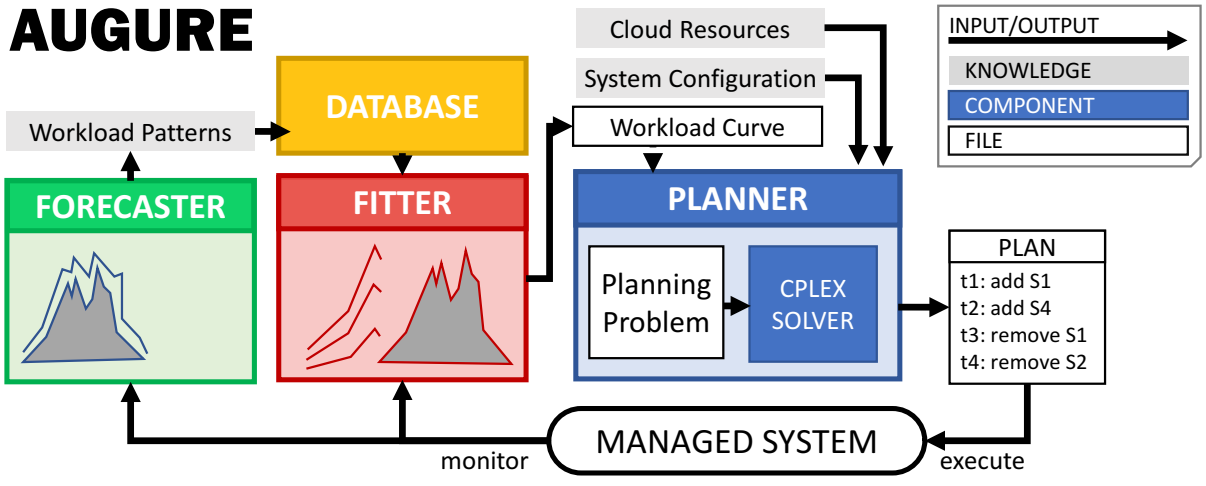

Figure 4.2: Approach Overview

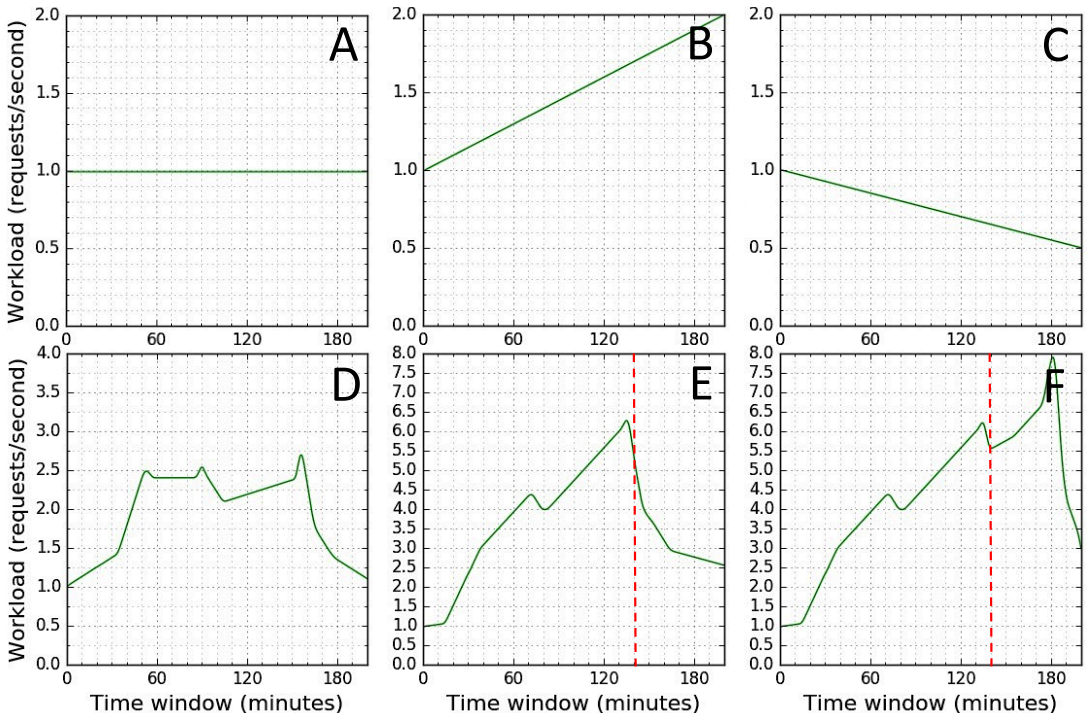

Figure 4.3: Approach: Workload patterns

4.2.1 Workload Patterns

A workload pattern denotes a set of workload curves sharing some characteristics. In Augure, these patterns are used to capture recurrent observed behaviors in the workload. A pattern is represented by a time series. Its data points should be regarded as normalized workload values to the value of the workload at time 0.

Fig. 4.3 illustrates several workload patterns. Pattern A denotes workloads that are constant in the foreseeable future. Patterns B and C denote scenarios where workload changes steadily. Patterns D, E, F denote more sophisticated realistic behavior, extracted directly from (The Internet Traffic Archive 2018). Interestingly, pattern E and F signal similar curves initially but deviate significantly from a point in time (see minute 140 in patterns E and F). It is worth noting that if only pattern A is available, the behavior of Augure is purely reactive, i.e. Augure assumes that the load will remain constant and only reacts to changes when they are observed.

The adoption of patterns that abstract from the initial workload value and focus only on the relative variation in a time horizon is justified by the observation that, while patterns are recurrent, values change according to user behavior. Thus, a workload curve may be different depending on the day of the week (e.g. workload volumes may be higher on weekends).

4.2.2 Pattern Extraction

Augure keeps a database of relevant workload patterns. Some patterns are pre-loaded in the system. Other patterns may be explicitly added to the system based on knowledge obtained elsewhere (e.g., leveraging the expertise of human operators). Also, patterns can be learned automatically at run-time. Each pattern is stored with a number of features that may subsequently help to match a specific workload behavior with the known patterns. *Contextual features* can be temporal (day of the week, season, holidays), trends (increasing or decreasing slopes), variances (burstiness levels), or information on sporadic events (team popularity in a football match).

The current prototype of Augure uses patterns that are captured offline (i.e. manually input by the operator) and patterns that are captured online, relying on a workload classification and forecasting approach proposed in (Herbst, Huber, Kounev, & Amrehn 2014). These techniques continuously provide a time series of point forecasts of the workload intensity with confidence intervals and accuracy metrics in configurable intervals. These forecasts have been normalized to the value of the workload at time 0 and categorized by comparing to the patterns already in the database. New patterns are added to the database when the current forecast does not match any of the existing patterns.

4.2.3 Fitting and Prediction

This task aims at predicting accurately the workload values in a given horizon, based on known patterns and the observed behavior during execution. It is divided into two sub-tasks:

Fitting, the first sub-task, tries to match the current workload behavior with one of the known patterns. Augure uses both the shape of the pattern and the contextual features associated to it to match the current execution with one of the patterns in the database. Historical data observed during a fixed interval (e.g. one hour) is compared to each existing pattern. Patterns are ranked by how similar they are to currently observed behavior, using standard deviation error weighted to the relative temporal past distance. The fitting process always returns at least one pattern. If no obvious candidate is found, pattern A is used. If several patterns see similar initial behavior, fitting will single out one of them, only when a significant deviations occurs (see patterns E and F, which deviate at minute 140).

De-normalization, the second sub-task, predicts the workload values. Augure creates a prediction of the workload by de-normalizing the pattern using the specific values observed in the current execution. For instance, assume that pattern B is selected after a one hour fitting interval. If at minute 60 the workload is 1300 req/s, then the predictor will estimate that the workload would increase up to a 2000 req/s by the end of the prediction window (minute 200).

The system is monitored periodically to confirm that the observed values are consistent with the prediction. When the monitored workload is within the confidence levels, the pre-computed plan continues to be executed. Whenever the monitored workload deviates out of confidence levels, the predicted values are adjusted to the observed behavior and a new plan is computed. If after a deviation there exists another pattern that matches better the observed behavior, such pattern is selected, de-normalized to the monitored values, and a new plan is computed.

4.2.4 Planning

The planning task takes the current configuration of the system, the selected workload prediction, and a list of resource types, to compute a sequence of adaptations that must be performed to meet the workload demand. The current version of Augure only considers two types of adaptations: activate or deactivate a server of a given type. Therefore, a plan is a sequence of activation and deactivation actions to be executed at particular time instants that deal with the expected increase and decrease in the workload. From a given initial configuration, the goal of planning is to find the best set of mixed resources and the best sequence of actions that minimizes a cost function, based on the assumption that the workload will follow the prediction. To this goal, Augure encodes the planning problem using a linear programming model and employs an off-the-shelf constraint solver, CPLEX (IBM 2016), to find an optimal solution. Similar techniques have been used in the past for short-term decisions in scenarios that do not consider a degradation impact of migrations (Srirama & Ostovar 2014).

The cost function considers both the direct costs associated to the active (rented) servers and the indirect costs incurred when clients are not served timely. In order to estimate these costs, Augure relies on a model of the servers and their operation. A server can be in different states: inactive, start-up, warm-up, active, and cool-down, before going back to inactive. The times it take to warm up and cool down a server are specific to the application and the cloud provider; these times must be estimated for each application (offline). The capacity and the

cost of a server depend on the server type and its state. An inactive server of any type has 0 capacity and 0 cost. The activation of an inactive server puts it on start-up state for some time, then it goes through a warm-up state and when warm up is over the server becomes active. The graceful decommission of the server requires the server to go through a cool-down state before becoming inactive again. The capacity of a server x in an active state, is denoted by p_x (in terms of requests per second) while the degraded capacity of a server x in the warm-up or cool-down state is denoted by d_x. Negative values of d_x can be used to capture the overhead that such server may induce on other servers due to state-transfer or other initialization or deactivation tasks. Servers that are not inactive have a cost that, in most cloud providers, is often charged in fixed billing periods (e.g. 60 minutes in Amazon EC2).

The cost function is encoded as follows. The direct cost associated to a server x is represented by c_x; this accounts for how much it is charged by the cloud provider when a server is requested. These costs are incurred every billing period and, once a server is requested, releasing it before the end of the next billing period brings no savings. Indirect costs are captured as a penalty l payed due to requests that are served slower or may even be dropped when the installed capacity is not enough to sustain the workload. This cost at a given point y is assumed to be proportional to the gap between the required capacity at that point (denoted by w_y) and the installed capacity given by the sum of the capacity of all servers that are not inactive. The workload demand is assumed to be served linearly with respect to the installed processing capacity, measured in number of processing units.

The encoding of the problem using linear programming uses three collections of boolean variables: the billing points $bill_{xy}$ indicate whether a server x is requested or renewed by the cloud provider at point y, the $active_{xy}$ indicates whether the server x is active at point y (i.e., serving some load), and $conf_{xy}$ indicates whether the server x is at in the warm-up or cool-down state at point y (i.e., state migration is going on). The planning window has duration $r + t + g$, where r is the maximum server start-up time (including warm-up time), t is the time window of the workload curve, and g is the billing period, measured in time-steps of τ minutes. Hence, the time instant that corresponds to a point y is $y \times \tau$ minutes. Under these assumptions, the cost function used by Augure is captured in Equation 4.1.

$$\sum_{y=0}^{r+t+g} (\sum_{x=1}^{s} c_x.bill_{xy} + l.(w_y - \sum_{x=1}^{s}(p_x.active_{xy} + d_x.conf_{xy}))) \tag{4.1}$$

4.2.5 Execution

The execution module takes the plan as input and schedules all the adaptations in the plan, from the current time until the end of the reassessment period. This involves requesting the activation of new servers or the decommissioning of active servers. It is worth mentioning that, while the current prototype is targeted at supporting horizontal scaling, the system may be extended to consider vertical scaling as well.

4.2.6 Continuous Re-evaluation

All the tasks above are executed periodically. New patterns can be discovered and added to the database in any cycle. Similarly, the selected pattern and the predicted values of the workload are re-assessed periodically, to take into account the real evolution of the workload. As a consequence, a more suitable pattern and more accurate values may be selected based on the data observed at run-time. If an updated prediction of the workload curve exists, planning is re-executed taking as input the current configuration of the system and the updated prediction. Finally, the most current plan is scanned and the scheduled actions are prompted.

It is worth noting that, in the absence of learned patterns, Augure behaves as a purely reactive controller, resorting by default to pattern A. Augure assumes that the load remains constant throughout time and, therefore, only reacts to changes when detected (at the end of a monitoring period).

4.3 Case Study

To evaluate Augure, let's consider a scenario with general purpose instances from Amazon, shown in Table 4.1. The processing capacity and the cost of the servers are defined by two units: $p = 50$ req/s. (equivalent to 1CPU) and $c = 0.006$ \$/hr. Realistic workload traces and patterns have been extracted directly from (The Internet Traffic Archive 2018)(Arlitt & Jin 2000). In particular, the case study concentrates on workload traces that match patterns D, E, and F (see Fig. 4.3). For the application, the server activation time (including server boot-up, configuration, and state migrations) was set to 12 minutes and the state migrations to 3 minutes.

Table 4.1: Cloud Resources: Amazon EC2 Instances - t2

Instance Size	CPU	Cost [\$/hr.]	P [p]	C [c]
t2.micro	2	0.012	2	2
t2.small	4	0.023	4	4
t2.medium	8	0.047	8	8

4.3.1 Testbeds

Augure was evaluated with two different testbeds:

Simulator: The discrete simulator implements a idealized managed system where the capacity of the system matches exactly the sum of the capacity of the machines that are active at a given time. Simulators are used to evaluate the controller's performance in isolation and without external noise.

E-Store: E-Store is an elastic in-memory database system designed for online transaction processing applications (Taft, Mansour, Serafini, Duggan, Elmore, Aboulnaga, Pavlo, & Stonebraker 2014). E-store supports high/low partitioning granularity for hot/cold tuples and employs live migration to distribute data among servers and achieve load balancing even in the face of skewed access patterns (Elmore, Arora, Taft, Pavlo, Agrawal, & El Abbadi 2015). An important aspect of E-Store is that adding/removing replicas involves an expensive data migration procedure that has a significant impact on the system.

4.3.2 Controllers

Four different controllers were used to investigate the effects of proactive scaling and heterogeneous resources:

Reactive (Baseline): E-Store is natively controlled using reactive techniques akin to those offered by cloud providers. The controller adjusts a pool of equally-sized servers. Servers are added/removed from the pool (one at a time) when the CPU utilization exceeds a pre-defined threshold. In the presented experiments, a homogeneous pool of *t2.medium* servers was used. One new server is added when the average CPU utilization is higher than 80% (6 minutes monitoring) and one server is removed when the average CPU utilization is lower than 25% (12 minutes monitoring).

Vadara: Vadara (Loff & Garcia 2014) combines greedy heuristics with short-term predictions to scale a pool of homogeneous resources. Vadara predicts workload variations in the next monitoring cycle and reacts accordingly, activating the necessary resources in advance. Also, Vadara does not deactivate servers immediately when the load drops; instead, it leaves servers in an idle pool and reuses them if the workload rises again before the rented time expires. In the experiments, a homogeneous pool of *t2.medium* servers is used. Vadara was fed with the prediction of the workload curve and the next monitoring cycle was fixed to be equal to the server activation time.

Vadara+: Vadara+ is an extension of Vadara that can manage heterogeneous resources. Vadara+ was developed as a contribution of this work. It uses a prediction of the workload variation in the next monitoring cycle to estimate the number of processing units needed in the near future. It employs *first fit decreasing* (FFD) heuristics to compute a combination of mixed server types that fits the workload in the next monitoring cycle and adapts the pool greedily. In the experiments, a heterogeneous pool of servers (*t2.micro*, *t2.small*, and *t2.medium*) was used. The next monitoring cycle was fixed to be equal to the server activation time.

Augure: Augure combines workload predictions and a long term horizon to find a schedule of actions to control a pool of heterogeneous resources. In the experiments, a heterogeneous pool of *t2.micro*, *t2.small*, and *t2.medium* servers is used and Augure is customized with $\tau = 3$ minutes and a monitoring cycle equal to the server activation time.

4.4 Evaluation

An extensive evaluation of Augure is presented in this section. Simulations and a real system implementations are used to assess various aspects of the planning approach and to compare the performance of different scaling techniques.

4.4.1 Planning Time

The simulator was used to measure how long it takes the planner component of Augure to compute an adaptation plan as a function of the problem size. The size of the problem, i.e. the number of constraints in the linear programming model, depends on two parameters: the "length" of the temporal window t measured in time-steps and the number of resources in the server pool. Augure's scalability was tested against different values of the time-step τ, thus changing the number of samples. For a fixed value of τ, the intensity of the workload curve was modified; the number of server instances needed to meet the workload varies accordingly.

Fig. 4.4 shows how the search time increases with the number of resources for each value of τ. Depending on the workload intensity, the value of the time-step can be selected to guarantee that search times are tamed. In average, CPLEX solves a large problem (around 30 thousand constraints) in less than 5 minutes; a positive result when compared to the monitoring and activation times. For the workload traces used in this evaluation, the number of instances ranges from 5 to 12, making it acceptable to fix $\tau = 3$ minutes.

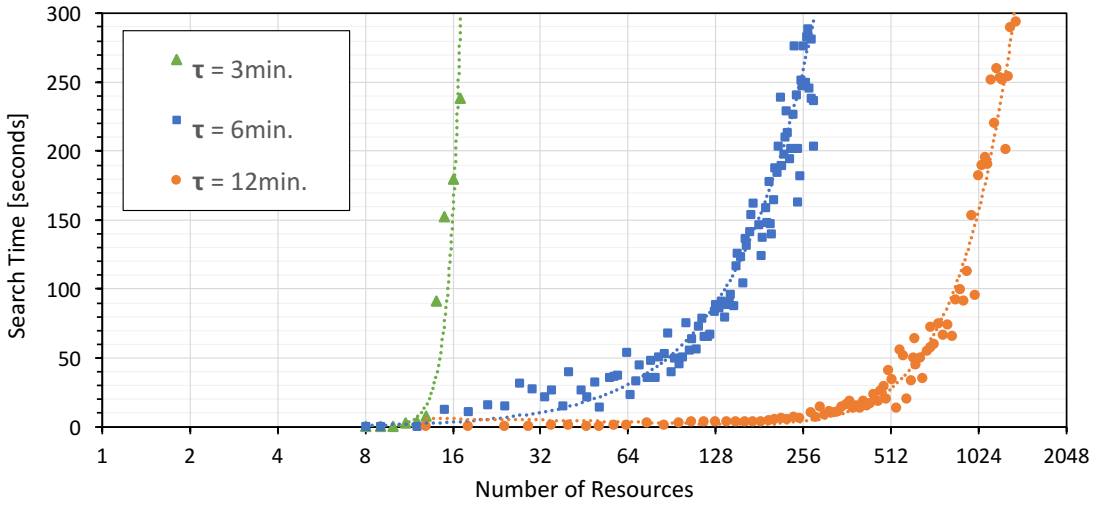

Figure 4.4: Scalability: Augure's planning time against number of resources

4.4.2 Fitting and Prediction

The simulator was used to evaluate Augure's ability to fit the current workload to a pattern in the database and the impact of miss-fitting and workload deviations on its ability to meet SLOs. The system was exposed to deviations in a workload matching pattern E (see Fig. 4.3).

Fitting and prediction results are depicted in Fig. 4.5. Augure is able to accurately perform the fitting task when the workload matches exactly one of the known patterns (Subfig. 4.5.1). When the workload values deviate significantly from the predicted curve (e.g. at time 84 in Subfig. 4.5.2 there is an abrupt surge), Augure readjusts its prediction successfully. Similarly, when the workload curve deviates from the selected pattern (e.g. at time 144 in Subfig. 4.5.3 the load starts following pattern F, instead of E), Augure is able to recognize the current pattern promptly. In such situations, Augure's miss-fit induces SLO violations for less than 10% of requests during the immediate cycle after the deviation up until the time when the effects of the updated plan are observed.

Also, Augure was evaluated in a scenario where it must resort back to pattern A in every cycle, behaving as a reactive system. In such case, SLO violations cannot be avoided, resulting in 10% to 50% of unserved requests per cycle (Subfig. 4.5.4). Results show that knowledge of workload patterns, even if not fully accurate, brings significant benefits.

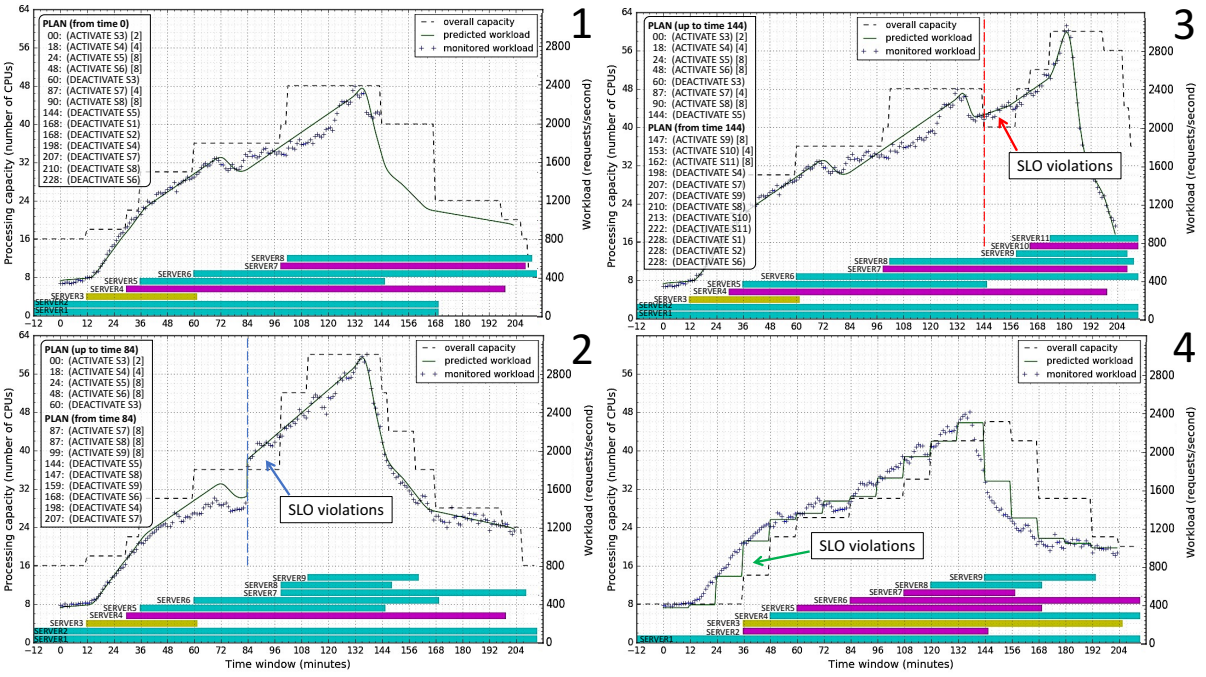

Figure 4.5: Responsiveness: Augure's fitting and miss-fitting effects on system performance

4.4.3 Cost and Reconfiguration Impacts

The simulator was used to assess the impact of reconfiguration actions. The system was exposed to a workload curve matching pattern D and proactive controllers were compared.

Let's consider three scenarios to capture the impact of reconfiguration actions on the system: local, global, and stop+go. **Local:** implies that activated servers have a percentage of their steady-state performance during warm-up/cool-down periods, but do not degrade the overall performance. **Global:** captures a scenario in which all active servers see performance degradation when one of the servers is warming up or cooling down. The overall capacity is considered to degrade with a factor d_x of four times the compound capacity of all servers in warm-up/cool-down state. **Stop+Go:** captures a scenario in which the system temporarily stops operating when a server is warming-up or cooling down; during these intervals, the processing capacity is null and all requests are dropped. For this simulation, the impact of reconfiguration is studied in scenarios with system degradation: *global* and *stop+go*. The indirect costs due to SLO violations were computed using the cost function with a loss factor $l = 0.001$ \$/min multiplied by the lacking capacity at each time-step.

Table 4.2 shows the bill (in cost units [c] and dollars [\$]) and the reconfiguration impacts (as Service Level Objective Violations SLOV and losses in dollars [\$]) for Vadara (V), Vadara+ (V+) and Augure (A). Vadara+ reaches a better fit for the workload curve using mixed resources and incurs lower bills than Vadara[*medium*]. Vadara generates less reconfiguration actions when using larger servers, which translates into fewer violations and monetary losses. In this particular setting, Vadara[*medium*] outperforms Vadara+ in terms of overall cost. In comparison to Vadara and Vadara+, Augure is able to reduce the number of reconfigurations by 20% (homogeneous) to 70% (heterogeneous). Augure[*mixed*] plans are also cheaper both in billed price and in monetary losses; a cost reduction up to 35%. As expected, *stop+go* inflicts more violations to the SLO for all three controllers, since it stops all service at every reconfiguration step.

Simulations confirm that greedy controllers that use short-term predictions may trigger many reconfigurations and, in scenarios with degrading impacts, this translates into monetary losses. Also, controllers that manage mixed resources find a better fit for the workload curve, reducing over-provisioning and incurring lower bills. Augure's planning on long-term horizons and mixed resources is able to derive plans that lower the billed costs while mitigating the impact of reconfigurations on the system performance.

Table 4.2: Cost Reductions: Vadara (V), Vadara+ (V+), Augure (A)

Controller [*server*]	Bill [c]	Bill [$]	Global SLOV	Global Loss[$]	Stop+Go SLOV	Stop+Go Loss[$]
V[*micro*]	82	0.4834	207	2.4840	269	3.2280
V[*small*]	88	0.5192	181	2.1720	205	2.4600
V[*medium*]	96	0.5664	159	1.9080	159	1.9080
V+[*mixed*]	84	0.4956	179	2.1480	241	2.8920
A[*micro*]	80	0.4720	165	1.9800	467	5.6040
A[*small*]	84	0.4956	178	2.1360	319	3.8280
A[*medium*]	96	0.5664	181	2.1720	205	2.4600
A[*mixed*]	82	0.4838	157	1.8840	232	2.7840

4.4.4 Live Reconfiguration in E-Store

In this subsection, Augure controller is evaluated as a solution to live reconfiguration of a transactional database management system, E-Store. The performance of the controllers on this system is assessed comparing the billed prices and the service degradation measured by two indicators: throughput and 50% latency. E-Store was deployed locally with a database of 60 million tuples that are each 1KB (60GB in total). Extensive experiments were conducted using large data-sets with uniform access patterns. All the experiments were conducted in a cluster of 20 Linux machines connected by a 10Gb switch. Each node has two Intel Xeon quad-core processors running at 2.13 GHz with 40GB of RAM. Servers are virtual machines with pre-configured capacities, equivalent to server types presented previously (see Table 4.1).

The system was exposed to workload traces matching pattern D. To generate transactional workloads, the standard YCSB workload generator (Cooper, Silberstein, Tam, Ramakrishnan, & Sears 2010) was augmented, such that the volume of transactions could vary over time. YCSB was configured to execute 85% read-only and 15% update transactions.

Before running the experiments on the real deployment, simulations were used to estimate the performance of the four controllers: Reactive, Vadara, Vadara+, and Augure. Vadara[*medium*] was selected since this is the best scenario for Vadara, as mentioned before. For this setting, the simulations indicated that both Augure and Vadara use seven servers and four activation actions. Unfortunately, Vadara+ requires ten servers and twelve activation actions. As noted above, controllers that combine greedy decisions with mixed resources trigger more56 reconfiguration actions. Given the poor performance of Vadara+ in this particular setting, this controller was excluded from the set of live experiments with the real testbed.

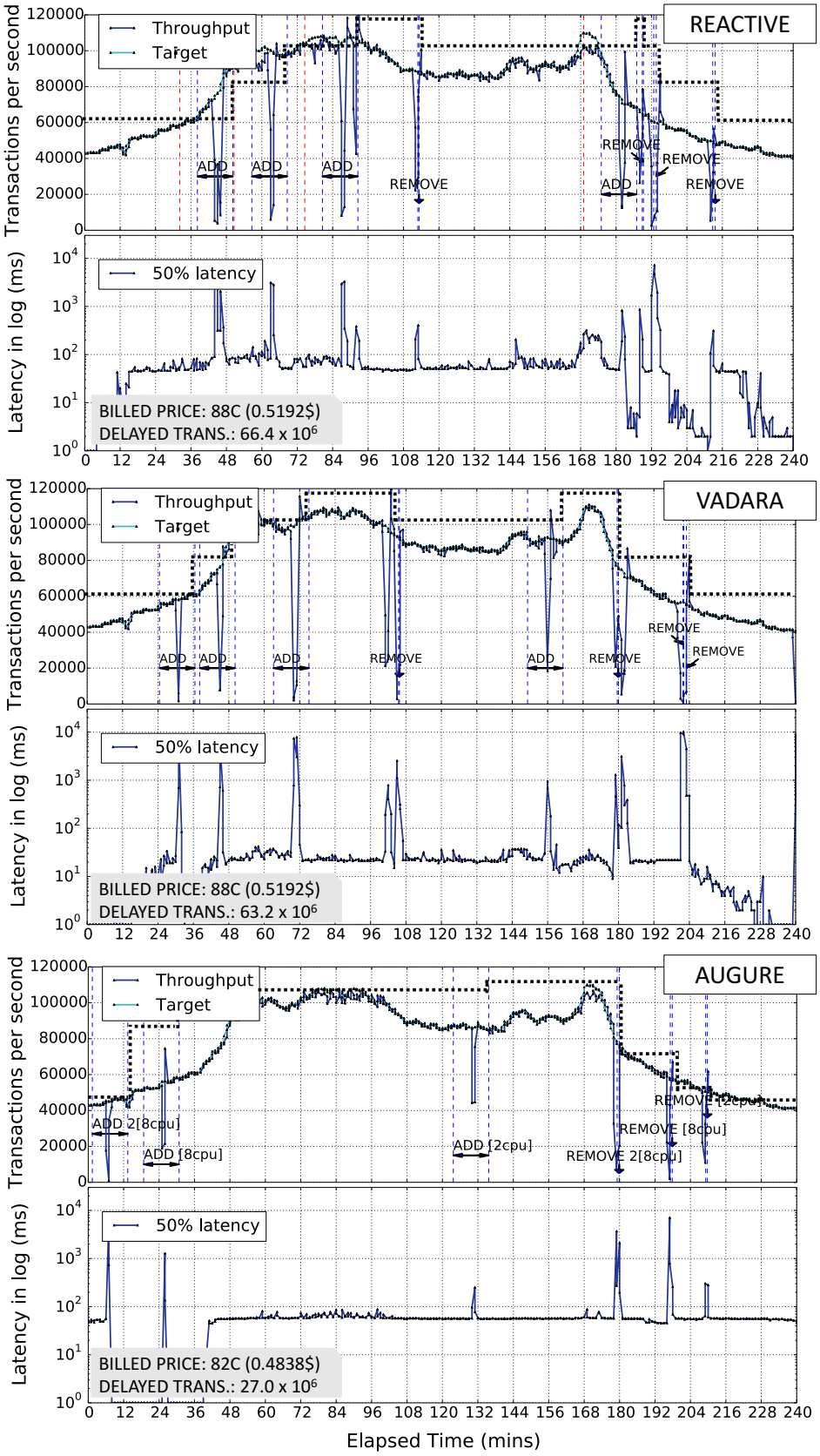

Figure 4.6: Scaling and Reconfiguration Impacts: E-Store live reconfiguration impact on throughput and latency. Controller comparison: Reactive, Vadara and Augure

Fig. 4.6 shows the results for Reactive, Vadara, and Augure in the live deployment of E-Store with pattern D. Each sub-figure corresponds to one controller and shows the service degradation measured by two indicators: throughput (in transactions per second) and 50% latency (in milliseconds). Each reconfiguration action induces a spike decrease in the measured throughput (in comparison to the target throughput), as well as a spike increase in the 50% latency. Every delayed transaction over 100ms is counted as an SLO violation. For each sub-figure, it is presented (low left corner) the total billed price by the cloud provider (in cost units c and dollars \$) and the total number of delayed transactions that violate the SLOs.

Results are interesting. Compared to the reactive approach, Vadara offers the same billed price but is able to decrease the number of delayed transactions by 5% (the reactive controller misses more transactions due to monitoring and activation delays). Augure offers significantly better results when compared to Vadara, reducing the billed price by 7%. More importantly, Augure reduces the number of reconfigurations and their impact on the system. It mitigates delayed transactions by 58% with respect to Vadara and by 60% with respect to Reactive.

4.4.5 Discussion

Augure resorts to a linear programming model to decide a schedule of scaling actions to accommodate resources such that the system capacity meets the (predicted) workload curve. Even if not explored in the proposal, a linear programming model is ideal to capture system requirements in the form of constraints to the adaptation. With regards to the optimization of the system performance and minimization of operational costs, Augure acts in two ways: (1) it imposes that the computational capacity must always be able to meet the (predicted) workload demand, to ensure that the throughput is maintained at acceptable levels; and (2) it optimizes a cost function that considers both the cost billed by the cloud provider due to resource consumption and the monetary penalty caused by performance degradation due to reconfiguration actions. Both mechanisms are embedded in the linear programming model.

Augure is responsive enough to act quickly and positively to environmental changes. In fact, its proactive nature makes so that it is able to act even before the (predicted) changes in the workload curve take place. Additionally, through fitting and prediction, Augure can readjust its plans whenever the workload curve deviates from the predicted pattern. At its worst, Augure behaves as a reactive controller, when no prediction is available. When predictions are inaccurate

and violations of the service level objectives are unavoidable, fast planning ensures that a new scaling solution is found before the next monitoring cycle.

Augure's planning is empowered by linear programming. The linear programming model encodes a simplified (and inflexible) description of the scheduling problem. This is beneficial in that constraint planners can solve a model with thousands of constraints in few minutes, making it highly scalable. Otherwise, languages and tools from AI temporal planning are unable to handle problems with long temporal horizons and many combinations of machine types.

4.5 Related Work

The solution presented above is based on the assumption that temporal patterns exist and can be effectively extracted, using techniques similar to the ones proposed in (Herbst, Huber, Kounev, & Amrehn 2014). In that proposal, a forecasting methodology that dynamically selects at run-time a suitable forecasting method for a given context is able to continuously provide time series of point forecasts of the workload intensity with confidence intervals and forecast accuracy metrics in configurable intervals and with controllable computational overhead during run-time. Many other temporal predictors (e.g. time series with moving average, auto-regression, exponential smoothing, or neural networks) and non-temporal predictors (e.g. support vector machines or decision trees) have been used in the past to estimate future workload values and resources (Qu, Calheiros, & Buyya 2016; Lorido-Botran, Miguel-Alonso, & Lozano 2014; Kim, Wang, Qi, & Humphrey 2016; Messias, Estrella, Ehlers, Santana, Santana, & Reiff-Marganiec 2016). Any technique able to generate long temporal workload predictions may be incorporated to the presented approach.

Proactive approaches for elastic scaling of resources in the cloud have been proposed in the past. However, most approaches only generate predictions of the workload in the short or medium term (Gong, Gu, & Wilkes 2010; Shen, Subbiah, Gu, & Wilkes 2011; Nguyen, Shen, Gu, Subbiah, & Wilkes 2013; Loff & Garcia 2014; Shariffdeen, Munasinghe, Bhathiya, Bandara, & Bandara 2016). Even solutions that have long-term predictions available (Jiang, Lu, Zhang, & Long 2013), resort to short-term greedy decision-making. All such solutions are unable to benefit from long-term scheduling that considers cloud provider's billing periods (one hour) to scale resources in a way that minimizes over-provisioning in time.

Other solutions exist that require some data reconfiguration after scaling resources. Previous approaches ignore the performance degradation caused by data reconfiguration when executing scaling actions (Lim, Babu, & Chase 2010; Shen, Subbiah, Gu, & Wilkes 2011; Didona, Romano, Peluso, & Quaglia 2014). To the best knowledge, only the SCADS Director (Trushkowsky, Bodík, Fox, Franklin, Jordan, & Patterson 2011) incorporates an estimation of the degradation due to data reconfiguration in the decision-making for elastic scaling of heterogeneous resources. Yet, similar to the other proposals, the SCADS Director does not benefit from a long-term vision to schedule the activation and termination of replicas proactively, such as to reduce the potential negative impacts of data reconfiguration.

While controllers that manage heterogeneous resources have been proposed before (Srirama & Ostovar 2014; Wang, Gupta, & Urgaonkar 2016; Verma, Gangadharan, Narendra, Ravi, Inamdar, Ramachandran, Calheiros, & Buyya 2016; Ma, Zhang, Zhang, & Zhang 2016), the controller presented in this chapter is more effective at combining machines of various sizes using planning to schedule them such as to minimize monetary costs in the long horizon. Previous approaches that provision resources in a greedy manner or with heuristics, without a long-term prediction of the workload curve, risk executing sub-optimal plans, instead.

4.6 Conclusions

The approach presented above aims at responding the question: How can temporal planning be used for the proactive reconfiguration (online) of interactive applications in cloud environments?

The proposed approach solves a temporal planning problem using a linear programming module and constraint solvers. This approach was successful at meeting the goals and addressing the limitation of previous work in that:

1. Augure is a proactive controller that used temporal planning to scale resources such as to accommodate the system capacity to the predicted workload. It enables a fitting and prediction mechanism to predict the future behavior of the workload, departing from a database of learned patterns. Plans generated online are executed before changes occur. It improves on previous work that are not responsive enough to react to fast changes.

2. Augure generates a model of the planning problem that considers a long temporal horizon, such as to generate a schedule of the activation and termination of machines that meets the capacity requirements while minimizing the monetary costs billed by the cloud provider. It improves on previous work that make short term decisions in a sub-optimal manner.

3. Augure optimizes a cost function that captures the monetary penalties due to performance degradation produced by data migrations and that also considers the (predicted) behavior of the environment to mitigate the negative impact of reconfigurations on the service. It improves on previous work that cannot account for the impact of reconfigurations.

Summary

This chapter has presented the implementation of an automated planning solution to support the proactive scaling and reconfiguration of cloud applications, and its evaluation via simulations and a prototype deployment in E-Store, an online transaction processing database system. The solution introduced Augure, a proactive decision-making tool for resource adaptation in cloud-enabled applications. Augure combines the benefits of long-term predictions of the workload curve and the use of heterogeneous resources to find adaptation plans that offer a good fitting of resources to the demand in the long horizon. Augure plans minimize the price billed by the cloud provider and mitigate the impact of reconfiguration on the system performance. Using off-the-shelf solvers, Augure can search the space of resource combinations and action schedules in less than 5 minutes. Augure recognizes workload behavior changes and adjusts the prediction promptly. Augure is able to tame the number of reconfigurations when compared to greedy proactive techniques (such as Vadara+), achieving better results and lower costs overall.

Publications

The work presented in this chapter has contributed to the following publication:

Augure: Proactive Reconfiguration of Cloud Applications using Heterogeneous Resources. R. Gil Martinez, Z. Li, A. Lopes, L. Rodrigues. Proceedings of the 16th IEEE International Symposium on Network Computing and Applications (NCA). Boston, USA. 2017

5 Workflow Executions with Spot Instances

This chapter introduces and evaluates a technique to automatically generate reactive policies for the deployment of workflow applications in the cloud, using spot instances. In particular, it addresses the question on how can *probabilistic planning* be used for the (offline) generation of policies that regulate the execution of workflows, using spot instances in cloud environments.

This chapter studies the use of automated planning as a tool to optimize the execution of deadline-constraint workflows in cloud environments. The proposed solution derives policies using models that capture the non-deterministic effects of deployment actions, to account for the probability of revocation of a spot instance. Planning is used to solve automatically generated Markov Decision Processes (MDP) that model the execution of the workflow and to explore the state space using well-known algorithms that lead to the optimal policy. Static policies are derived before the execution of a job (offline). These policies are then executed at run-time and guide the selection of deployment actions, depending on the occurrence of failures and the actual task completion times.

5.1 Motivation and Goals

Many multi-tier web and data processing jobs are represented as workflows, which capture the desired execution order of the tasks within a job and helps to reason abstractly about how to schedule such tasks in a deployment environment.

Workflow scheduling has been studied extensively in scenarios where resources are homogeneous and task completion times are predictable. In such scenarios, is is possible to find a schedule that minimizes the makespan and meets the timeliness requirements of the application. However, in heterogeneous scenarios, where several machine types are available and the task duration and price depend on the machine type, the search space becomes much larger and minimizing the makespan may no be longer the main objective. In fact, other schedules may exist

that still comply with the timeliness requirements and can be executed on cheaper deployments. Furthermore, very few works are prepared to deal with the uncertainty that is introduced by the use of revocable instances.

How to exploit the availability of revocable instances when planning the execution of workflows is a challenging problem. Specifically, it is planning problem that must resolve scheduling constraints, selection of machine sizes, and uncertainty. First, all tasks must be scheduled in a way that reduces cost but also ensures that the job finishes before its deadline. Second, machines must be chosen carefully taking in consideration the resources required by each task and their costs; one must decide both the size (e.g., small) and reliability (on-demand or spot instances) to minimize the expected execution cost. Third, the uncertainty introduced by performance variability and instance revocations must be accounted for to ensure the timely execution of a plan. To take full advantage of the opportunities offered by cloud providers, it is important to consider not only the execution time and the cost of each task, but also the probability of success; in particular, how likely it is that resources may be revoked.

The Importance of Planning: Since the problem of deciding machine types and time schedules for workflow executions in the cloud is complex, planning could be used to find the best solution.

To illustrate the importance of planning to find the best solution, a simplified scenario is shown in Fig. 5.1. Here, a workflow W is composed of five sequential tasks with equal fixed durations (1 time unit) and different resource requirements: T_1 to T_4 (1 CPU) and T_5 (16 CPU). The workflow must finish execution before a deadline $d_W = 6$ time units. To take advantage of heterogeneity, let us consider that two machine sizes are available: small S (1 CPU) and extra large X (16 CPU). Let's assume that each task is executed in a machine size matching their requirements (e.g. T_1 is executed in machine size S). Each machine can be rented on-demand or as a revocable instance. The price paid for revocable instance is assumed to be $\beta = 0.25$ times the price on-demand and the success probability is $p = 0.8$ (the probability of non being revoked in one time unit). To take advantage of free time slots, the application must decide which tasks to execute using on-demand or revocable instances, to ensure that all tasks finish successfully and the execution cost is minimized.

Figure 5.1 shows three different policies to assign machine types to tasks. Policy A selects

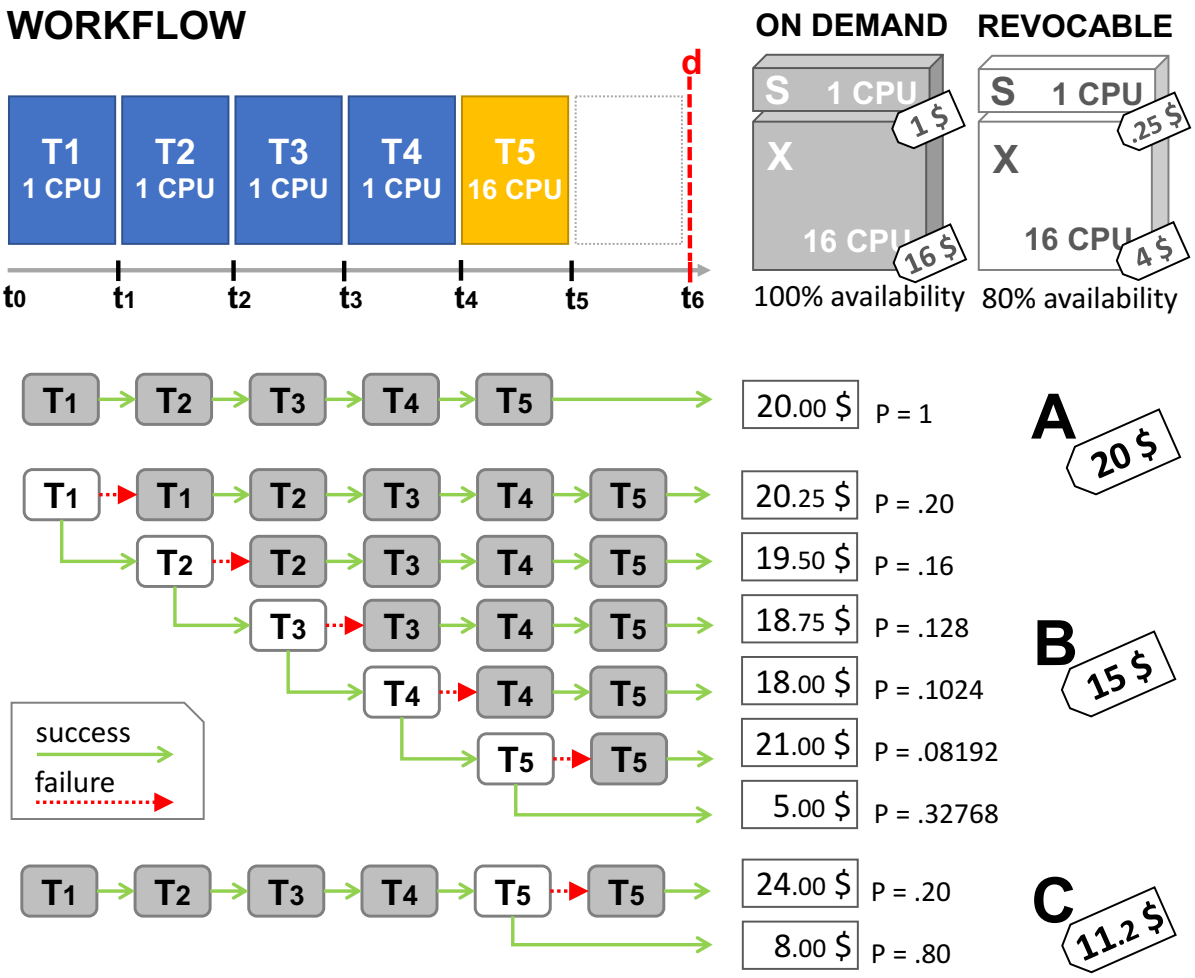

Figure 5.1: Motivation: Workflow execution using three different policies: (A) on-demand only, (B) revocable greedy, and (C) planning

on-demand instances only. The solution has the highest possible cost and under-utilizes the available time. Policy B assigns the cheapest revocable instance to each task in each time step (greedily) and resorts back to on-demand machines only when the time left does not allow for task failures. This policy ideally follows an execution path in which all tasks are completed using revocable instances, resulting in the cheapest cost (5$); yet, this solution occurs only with probability 0.32768. The expected cost of policy B, considering all possible solutions is instead of 15$. Policy C is derived using planning, i.e. exploring valid execution paths and comparing them to decide the best choice at each step. For this workflow, the optimal policy assigns a revocable machine to the execution the most expensive task: if the task succeeds, the cost is 8$ with high probability of 0.8. The expected cost of the solution is 11.2$, which is 25% cheaper than the one derived using greedy heuristics.

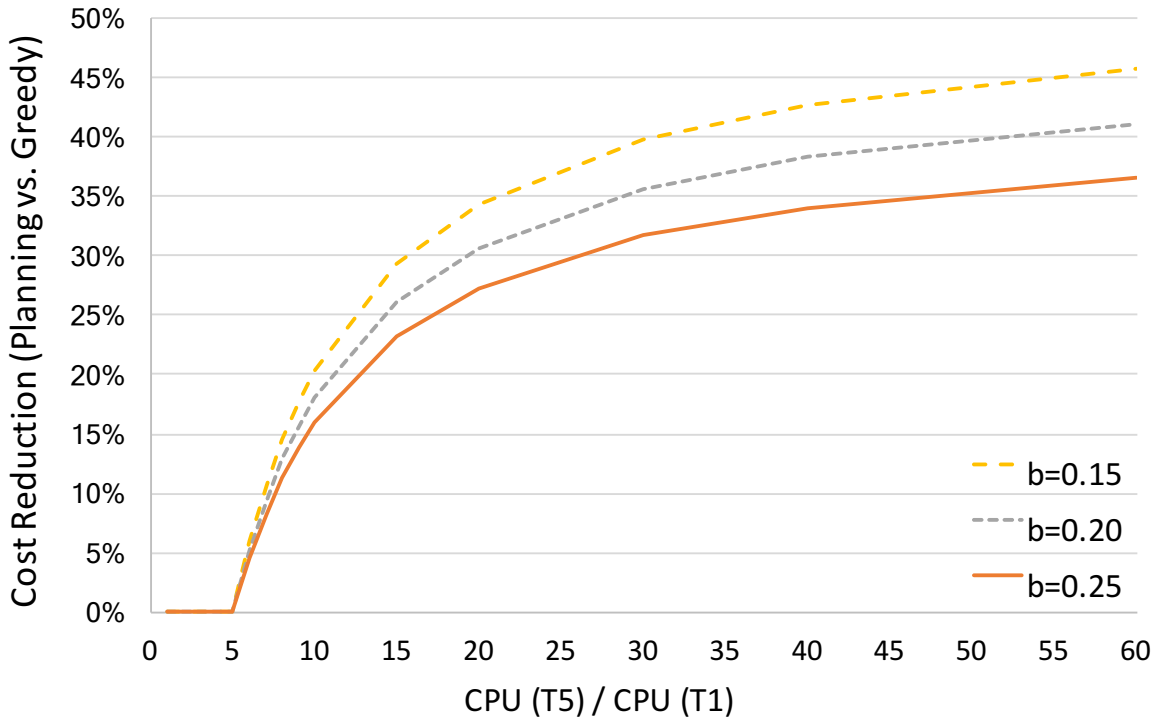

Figure 5.2: Motivation: Costs reductions of planned policies (w.r.t. greedy heuristics) against the cost ratio between the most expensive and cheapest task

The example above considers a small workflow with a limited number of machine types. In real life, scenarios are more complex and the space of solutions is larger. The benefits of using planning vary depending on the specifics of the worfklow graph and the variety of machine types.

As a generalization of the workflow described before, it is possible study the benefits in cost reduction of planning (w.r.t. greedy solutions) for different rates of prices of revocable instances, using the same workflow graph while varying the ratio of resources required by T_5 with respect to $\{T_1, ..., T_4\}$. Fig. 5.2 shows the curve for three prices of revocable instances: $\beta = 0.15, 0.20, 0.25$ times the price on-demand, with success probability $p = 0.8$. Planning is always able to find the cheapest solution and can reduce costs up to 40%.

Goal: To use probabilistic and temporal planning for the (offline) generation of execution policies for the deployment of workflow applications in cloud environments. In particular: (1) to take full advantage of the cost reduction opportunities introduced by spot instances; and (2) to generate policies that surmount the uncertainty caused by instance revocation and guarantee the reliable execution of workflows with timely requirements.

5.2 Problem Formulation

Workflow: A workflow $W = \langle \mathcal{T}, G_W, d_W \rangle$ is defined by a set of tasks $\mathcal{T}$, a labelled directed acyclic graph G_W whose nodes are labelled by tasks in $\mathcal{T}$ and a deadline d_W. The notation $i : T$ is used to denote that node i is labelled by task T. Every task in $\mathcal{T}$ is required to label at least one node but a task can be used to label different nodes. A (directed) edge in the graph, from $i : T$ to $j : T'$ represents dependency of data from task T to task T'. In this case, $i : T$ is said to be a parent of $j : T'$ and $j : T'$ is said to be a child task of $i : T$. A child cannot be executed until all of its parents are completed. Each task T has a constraint X_T on the resources required for its execution (e.g. minimum number of processing cores). The deadline d_W represents the time limit for the execution of the workflow.

Resources: Cloud providers offer virtual machines for different applications: compute optimized, memory optimized, etc. A virtual machine can be allocated different amounts of computational resources, pre-defined by the cloud provider, such that there are different machine sizes. Virtual machines can also be rented in different price markets, as on-demand instances or revocable instances. On-demand instances have a fixed price per time unit and reliability guarantees to the user. Revocable instances have a dynamic price that varies with the bidding market and are revoked when the price of the instance exceeds the price bid by the user. The ratio between the price paid for a revocable instance and the price of the instance rented on-demand is β. The entire set of virtual machines available to the workflow application is denoted by V. A virtual machine $v \in V$ is defined by a tuple $\langle cpu_v, mem_v, c_v(t), p_v(t) \rangle$, where: cpu_v is the number of processing cores, mem_v is the memory size, $c_v(t)$ is the cost of an instance rented for t time units (as defined by the price market[1]), $p_v(t)$ is the probability of the rented instance not being revoked during the first t time units.

Task Execution: Since tasks have different resource requirements, they could be executed in different virtual machine types and, hence, might have different execution times. Let duration[2] $d(T, v)$ be the maximum execution time required to complete task T in a virtual machine v,

[1] A billing policy per second is assumed.
[2] $d(T, v)$ is expressed in time units of 1 second.

taking into account the launching time and the performance variability[3]. These durations allow us to derive the expected cost[4] $c(T, v)$ and success probability $p(T, v)$ of executing a task T in an instance v: $c(T, v) = c_v(d(T, v)))$ and $p(T, v) = p_v(d(T, v)))$[5]. Task failures are assumed to occur only due to instance revocation. Also, tasks are atomic and must be executed again if they fail. To control the number of times each $i : T$ can be re-executed due to a task failure, the bound y is introduced. If a task $i : T$ fails y consecutive times using revocable machines, $i : T$ must be re-executed using on-demand machines. To ensure that the workflow W can be executed successfully using the machines in V, it must be verified that the minimum makespan $M(W, V) \leq d_W$. $M(W, V)$ is the time required for the execution of all nodes in G_W, when tasks use the fastest on-demand machine $v \in V$.[6]

User Preferences : When executing a workflow W, the application user may be interested in optimizing the execution to reduce monetary costs incurred by the utilization of virtual machines or to reduce the makespan. To account for user preferences, the parameter u is used to indicate a cost preference ($u = 1$) or a time preference ($u = 0$).[7]

Planning Problem: The execution of a workflow W must be planned, to decide what machine type to use for the execution of each tasks and when to schedule each action, such that the deadline is met and the performance is optimized. This planning problem has both temporal and probabilistic features. In this formulation, the *probabilistic planning* definition is combined with augmented states that include temporal variables.

Formally, the planning problem in defined in terms of a set of states S, an initial state $s_0 \in S$, a set of goal states $S_G \subseteq S$, a set of actions A, a subset of actions $A(s) \subseteq A$ applicable in each state s, a transition probability function $P(\cdot \mid s, a)$ for every action $a \in A(s)$, a reward function $R(\cdot \mid s, a)$ for every action $a \in A(s)$, and a sequence of steps L.

Specifically, assuming the enumeration $\{1 : T_1, ..., n : T_n\}$ of the nodes of G_W, the problem is defined by:

[3]Execution times are assumed to be fixed durations.
[4]$c(T, v)$ is expressed in cost units, defined with respect to the cheapest instance
[5]On-demand instances are assumed to have a success probability of 100%.
[6]It is assumed that for each task $T \in \mathcal{T}$ there exists at least one $v \in V$ with the required resources X_T.
[7]A combination of cost and time (e.g. as a weighted sum) could be easily incorporated in future work.

- S, the *set of states*, consists of all pairs of the form $s = (\langle b_1, ..., b_n \rangle, \langle t_1, ..., t_n \rangle)$, where b_i is a natural number ranging from 0 to $y+1$ representing the execution state of task T_i — 0 means waiting, any value in $\{1, ..., y\}$ means failed, and $y+1$ means successfully executed; while, t_i is a natural number ranging from 0 to d_W that captures the time evolution for the execution of $i : T_i$. Also, b_i and t_i are used as state variables and write $b_i(s)$ and $t_i(s)$ to refer to their values.

- s_0, the *initial state*, is such that $b_i(s_0) = 0$ and $t_i(s_0) = 0$, for $i = 1, ..., n$.

- S_G, the *set of goal states*, is the set of states s such that $b_i(s) = y + 1$ and $t_i \leq d_W$, for $i = 1, ..., n$.

- A, the *set of actions*, comprises the execution of each task in every virtual machine that has enough resources to execute it. If v is a virtual machine that satisfies X_{T_i}, then $a(i : T_i, v)$ is used, or a_{iv} for short, to denote the execution of $i : T_i$ in v.

- $A(s)$, the *set of actions applicable in each* $s \in S$, is defined as follows: $a_{iv} \in A(s)$ if and only if: (1) $i : T_i$ has not been successfully executed, i.e. $b_i(s) \neq y + 1$; (2) all parents of $i : T_i$ have been successfully executed, i.e. $b_j(s) = y+1$, if $j : T_j$ is a parent of $i : T_i$; (3) the time after the execution of $i : T_i$ does not exceed the deadline, i.e. $\max(\{t_i(s)\} \cup \{t_j(s) \mid j : T_j \text{ is a parent of } i : T_i\}) + d(T_i, v) \leq d_W$; and (4) the execution of $i : T_i$ has failed less than y times i.e. $b_i(s) < y$, if v is a revocable instance.

- $P(\cdot \mid s, a_{iv})$, the *transition probability function* for each $s \in S$ and $a_{iv} \in A(s)$, is such that $P(s'|s, a_{iv}) = 0$ if $t_i(s') \neq \max(\{t_i(s)\} \cup \{t_j(s) \mid j : T_j \text{ is a parent of } i : T_i\}) + d(T_i, v)$. Otherwise, $P(s'|s, a_{iv}) = p(T_i, v)$ if $b_i(s') = y + 1$ and $P(s'|s, a_{iv}) = 1 - p(T_i, v)$ if $b_i(s') = b_i(s) + 1$.

- $R(\cdot \mid s, a_{iv})$, the *reward function* for each $s \in S$ and $a_{iv} \in A(s)$, is such that $R(s'|s, a_{iv}) = R(a_{iv}) + R(s')$. $R(a_{iv}) = -c(T_i, v)$, if user preference is cost ($u = 1$). $R(a_{iv}) = -d(T_i, v)$, if user preference is time ($u = 0$). $R(s') = -R^{\infty}$, for a dead-end state $s' \notin S_G$ and $A(s') = \emptyset$; $R(s') = 0$, otherwise. R^{∞} is a large finite number.

- L is a sequence of natural numbers $1, 2, ..., L_{max}$ that represent the *decision steps* at which an action is executed. $L_{max} = n * (y + 1)$ is called the *horizon*.

The formulation of the planning problem corresponds to a finite-horizon Markov Decision Process (fh-MDP) of the form $\langle S, A, P, R, L\rangle$ (as defined by (Mausam & Kolobov 2012)). To make the presentation of the problem self-contained, below the key concepts regarding planning with MDPs are recalled (also following (Mausam & Kolobov 2012)):

An *execution history* $h_l = ((s_0, a_0), ..., (s_{l-1}, a_{l-1}), s_l)$ is a sequence of pairs of states the agent has visited and actions the agent has chosen in those states, at each step i, $0 \leq i \leq l-1$, plus the state visited in step l. The set of all possible execution histories is denoted by H. Note that, since it is assumed that every task in $\mathcal{T}$ can be executed in at least one machine in V and the minimum makespan is not greater than the deadline, there exists at least one execution history starting in the initial state s_0 and ending in a goal state.

A deterministic *history-dependent policy* $\pi_{h_l} : H \rightarrow A$ is a mapping that assigns to each $h_l \in H$ an action $a \in A$ (the action to be executed after the execution history h_l). If the decision is independent of the execution history h_l and depends only on the current state s and the number of steps l that led to s, the policy is *markovian.* A *markovian policy* $\pi : S \times L \rightarrow A$ is a mapping that assigns to each s reached in a step l (denoted by s_l) an action $a \in A(s)$.

The *value function* $V^{\pi}(s_l) = U(R_l^{\pi_{s_l}}, R_{l+1}^{\pi_{s_{l+1}}}, ...)$ of a markovian policy π is a utility function of a sequence of rewards $R_l^{\pi_{s_l}}, R_{l+1}^{\pi_{s_{l+1}}}, ...$; where $R_l^{\pi_{s_l}}$ is the reward obtained as a result of executing policy π from state s at step l. In MDPs with finite horizon L_{max}, the value function is defined by the expected linear additive utility, such that for all $s \in S$:

$$V^{\pi}(s_l) = \begin{cases} \mathbb{E}[\sum_{j=l}^{L_{max}} R_j^{\pi_{s_l}}] & 1 \leq l \leq L_{max} \\ 0 & l = L_{max} + 1 \end{cases}$$

The *optimal policy* π^* is such that the optimal value function V^*, the value function of π^*, dominates the value function of any other policy π, for all $h_l \in H$, i.e., $V^*(h_l) \geq V^{\pi}(h_l)$.

For fh-MDPs, the optimality principle ensures that:

(1) the optimal value function V^* exists, is markovian, and satisfies, for all $s \in S$ and $1 \leq l \leq L_{max}$:

$$V^*(s_l) = \max_{a \in A} \left[\sum_{s_{l+1} \in S} P(s_{l+1}|s_l, a)[R(s_{l+1}|s_l, a) + V^*(s_{l+1})] \right]$$

(2) the optimal policy π^* corresponding to V^* is deterministic markovian and satisfies, for all $s \in S$ and $1 \leq l \leq L_{max}$:

$$\pi^*(s_l) = arg\max_{a \in A} \left[\sum_{s_{l+1} \in S} P(s_{l+1}|s_l, a)[R(s_{l+1}|s_l, a) + V^*(s_{l+1})] \right]$$

The *optimal solution* of a fh-MDP is the optimal policy π^*. The objective of the planning problem defined by $\langle S, A, P, R, L \rangle$ is to find the optimal policy π^*.

An important observation is that the optimal policy π^* for the fh-MDP defined above is *stationary*, i.e. the decision is independent of the execution step l and depends only on the state s. This is because reachable states s contain enough information from which to derive the number of steps l to reach s. Here on, $\pi^*(s)$ is used instead of $\pi^*(s_l)$.

In addition, from the definition of the optimal value function V^* and the definition of the reward function R presented above, follows that:

(1) Any execution history $h_l(\pi^*)$ that result from the execution of the optimal policy π^* starting in s_0, ends in a goal state $s_l \in S_G$. This can be understood intuitively from the imposition of a penalty $-R^\infty$ to any execution history ending in a dead-end state $s_l \notin S_G$ and $A(s_l) = \emptyset$. Thus, when selecting π^*, any execution history ending in a goal state is always preferable to one ending in a dead-end state.

(2) the optimal policy π^* is such that $V^*(s_0)$ minimizes the expected linear additive cost (or duration, depending on the user preference) of the set of possible execution histories that result from the execution of π^*, starting in the initial state s_0 and terminating in s_l. This notion of optimality implies that any other policy results in a set of execution histories with a higher expected linear additive cost (or duration).

The *utility* of the optimal policy π^* is defined as $U(\pi^*) = V^*(s_0)$. Depending on the user preference, $U(\pi^*) = -C(\pi^*)$ or $U(\pi^*) = -D(\pi^*)$, where $C(\pi^*)$ and $D(\pi^*)$ denote the expected linear additive cost and duration, respectively.

5.3 Approach

In the previous subsection, the planning problem has been formulated as a fh-MDP of the form $\langle S, A, P, R, L \rangle$. The resolution of MDPs (and planning problems, in general) faces two main challenges: the state space explosion and the need for efficient algorithms. In the formulation, the state space explosion comes as a natural consequence of considering the execution of each task T_i in each virtual machine v, which defines a large set of reachable states $s \in S$ and applicable actions $a_{iv} \in A(s)$. The need for efficient algorithms comes for the desire to reduce the overhead of finding the optimal solution, both in terms of computational expenses and time.

In this subsection, an approach to planning the execution of workflows in cloud environments is presented. The approach consists of three stages: preparation, planning, and execution. In the *preparation* stage, the execution of all tasks is characterized, the workflow graph is reduced, and a reduced number of machines is pre-selected. In the *planning* stage, the MDP model is built and the value iteration algorithms is used to find the optimal policy. In the *execution* stage, the decisions dictated by the optimal policy are applied, as the controller evaluates the outcome of each action on-the-fly.

These stages are described below:

5.3.1 Preparation

Task Characterization: The execution of a task T in a virtual machine v is characterized by the mean CPU utilization $cpu_v(T)$, the peak memory utilization $mem_v(T)$ and the maximum execution time $d(T, v)$. A task characterization only exists if v has the minimum number of resources X_T required for the execution of T. This characterization can be obtained experimentally (by running all tasks on all possible machines) or by combining experimentation with machine learning to build a performance model for each task without exploring the complete configuration space (Alipourfard, Liu, Chen, Venkataraman, Yu, & Zhang 2017).

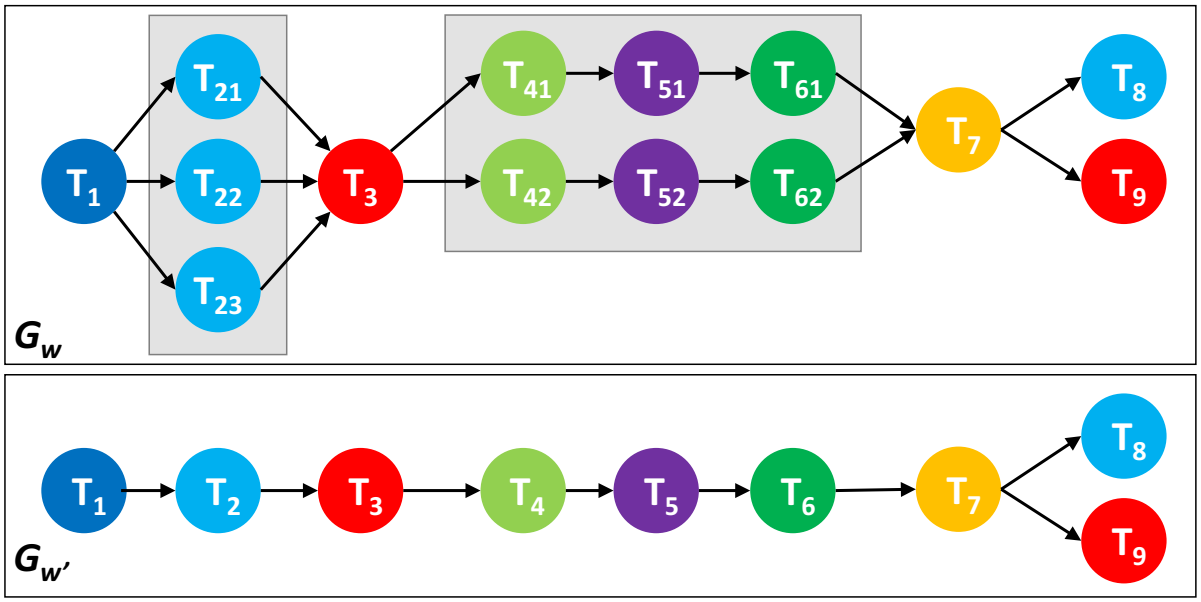

Figure 5.3: Approach: Workflow graph reduction

Workflow Graph Reduction: A workflow graph G_W containing concurrent pipelines is reduced to its minimal form, that is denoted as $G_{W'}$. A *pipeline* is defined as a sequence of nodes such that: (1) every node in this sequence has a single parent and a single child, (2) the child of every node but the last is a node also in the sequence, (3) the parent of every node but the first is a node also in the sequence. Concurrent pipelines are sets of pipelines with the same length, that are labelled with the same tasks (the i-th nodes of all pipelines in the set are labelled with the same task) and that have a common source and sink (the first nodes of all pipelines have the same parent and the last nodes have the same child). A task T_i executed in a set of concurrent pipelines is denoted by $T_{(i,j)}$, for $j = 1, ..., z_i$. The notation $T_{(i,j)}$ is used as a shorthand for $(i, j) : T_i$. Figure 5.3 presents an example where there are two sets of concurrent pipelines: $\{T_{(2,j)}\}$, for $j = 1, 2, 3$, and $\{(T_{(4,j)}, T_{(5,j)}, T_{(6,j)})\}$, for $j = 1, 2$.

Consider one set of z concurrent pipelines of length m of the form $\{((k, j) : T_k, ..., (k+m, j) : T_{k+m})\}$. This set can be collapsed into one pipeline of the form $\{k : T_k, ..., k + m : T_{k+m}\}$. The reduced workflow graph $G_{W'}$ is obtained by considering all sets of concurrent pipelines and collapsing each one into a unique pipeline as described before.

The idea of solving a MDP for the reduced workflow W' originates from a couple of observations regarding the policies of the MDP for workflow W.

(1) Consider two policies that only differ in the order they decide to execute tasks in concurrent pipelines (e.g., in Fig. 5.3, consider a state in which T_1 is completed but $T_{(2,1)}$, $T_{(2,2)}$ and $T_{(3,3)}$ have not been executed, a policy that chooses to execute $T_{(2,1)}$ and a policy that chooses to execute $T_{(2,3)}$, instead). Then, the utility for these policies is the same.

(2) Consider two actions $\pi_1(s) = a((i,j){:}T_i, v)$ and $\pi_1(s') = a((i,j'){:}T_i, v')$ defined by a policy π_1, that decides the execution of the task T_i in two concurrent nodes (i,j) and (i,j') in two machines v and v', when in two states s and s' such that $b_{(i,j)}(s) = b_{(i,j')}(s')$ and $t_{(i,j)}(s) = t_{(i,j')}(s')$. The policy π_2 that is obtained from π_1 by changing only the decision on s' to $\pi_2(s') = a((i,j'){:}T_i, v)$, has still the same utility.

These properties make it possible to derive a policy π for W from a policy π' for W' preserving optimality. Consider a reduced workflow W' has n' nodes. Let $F_i(s)$, for $i = 1, .., n'$, be the function that assigns to each state $s \in S$ (of W), the minimum number j_{min}, for $j_{min} = 1, ..., z_i$, such that $b_{i,j_{min}}(s) = \min\{b_{i,j}(s) \mid j = 1, ..., z_i\}$. Intuitively, the function F_i chooses a j_{min} where task $T_{i,j_{min}}$ has not been successfully executed, if there is any. If task $T_{i,j}$ has been successfully executed for all $j = 1, ..., z_i$, then F_i just chooses $j = 1$. These functions are used to define a mapping ψ from the states $s \in S$ to the states $\psi(s) \in \Psi$, where Ψ is the set of states of the MDP for W'. In particular, $\psi(s)$ is a state such that $b_i(\psi(s)) = b_{i,F_i(s)}(s)$ and $t_i(\psi(s)) = t_{i,F_i(s)}(s)$. The derived policy π is defined by $\pi(s) = a((i, F_i(s)) : T_i, v)$ iff. $\pi'(\psi(s)) = a(T_i, v)$. That is to say, for a set of concurrent pipelines, among the state variables defined for a task in concurrent nodes,the decision of π' is used considering only the state variables that refer to that task in the selected node.

Machine Pre-Selection: The set of all possible combinations of machines $v \in V$ and tasks $T_i \in G_w$, V_W, may contain machine types that are not good candidates for an optimal solution. Thus, a preliminary selection of machine types v' is selected for the execution of each task $T_i \in G_W$, defining the set V'_{T_i}. In particular, $v' \in V'_{T_i}$ if it satisfies the conditions: (1) v' has the minimum number of resources X_{T_i} required for the execution of T_i; (2) v' minimizes the expected cost of task T_i under equal duration i.e., $c(T_i, v') \leq c(T_i, v)$ if $d(T_i, v') = d(T_i, v)$, for any $v \in V$; (3) v' can execute task T_i without exceeding the deadline, i.e., $d(T_i, v') + \sum_{j \neq i} d_{min}(T_j, v) \leq d_W$, where $d_{min}(T_j, v)$ is the minimum duration for a task T_j, $j \neq i$, using machines $v \in V$. The set of pre-selected machines for W is a subset $V'_W \subseteq V_W$, defined as: $V'_W = V'_{T_1} \cup ... \cup V'_{T_n}$.

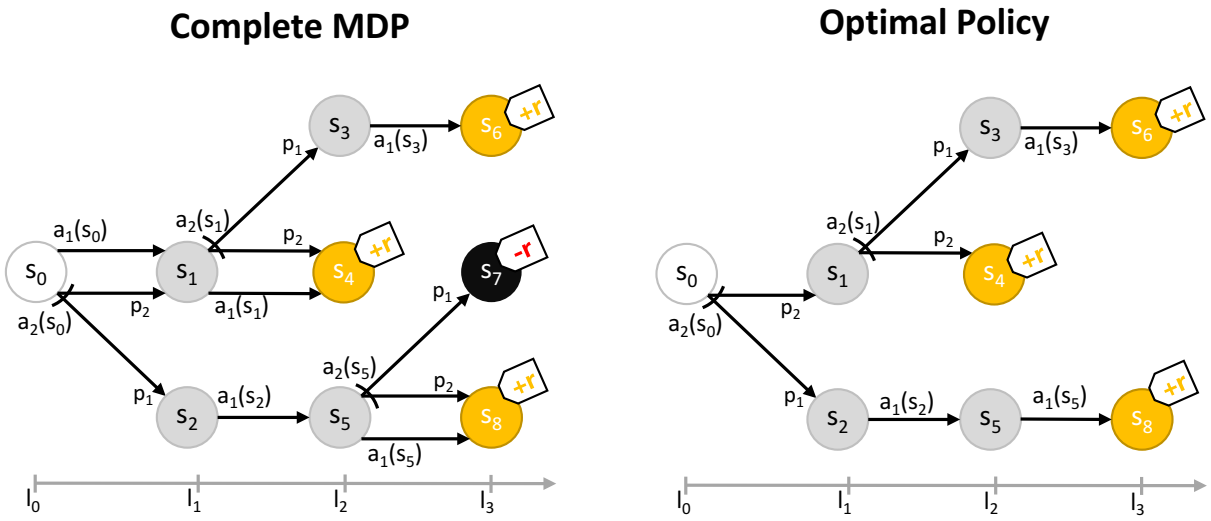

Figure 5.4: Approach: Planning with MDP

5.3.2 Planning

MDP Construction: The MDP construction takes as input the task characterization, the reduced workflow and the pre-selected machines. A MDP is constructed using the module formalism presented in (Kwiatkowska, Norman, & Parker 2011). The MDP model is constructed automatically using a MDP framework. The MDP framework represents the execution of all nodes in one module, where each action a_{iv} is represented with a tag $[iv]$, where i is the node number and v is the virtual machine type (e.g. small S, on-demand O or revocable Q).

Listing 5.1: Reactive Module Formalism for MDP Construction

```
module w
b1 :[0.. y+1]; b2 :[0.. y+1]; t1 :[0..  dw]; t2 :[0..  dw];
[1SO] b1<y+1 & t1+d1<dw -> 1.0: b1'=y+1 & t'=t+d1;
[1SQ] b1<y  & t1+d1<dw -> 0.8:  b1'= y+1 & t'=t+d1 +0.2: b1'=b1+1 & t'=t+d1;
[2SO] b2<y+1 & b1= y+1 & max(t1,t2)+d2<=dw -> 1.0: b2'=y+1 & t'=max(t1,t2)+d2;
[2SQ] b2<y  & b1=y+1 & max(t1,t2)+d2<=dw -> 0.8: b2'= y+1 & t'=max(t1,t2)+d2
                                                  +0.2: b2'=b2+1 & t'=max(t1,t2)+d2;
endmodule
rewards
[1SO] −c1SO; [1SQ] −c1SQ; [2SO] −c2SO; [2SQ] −c2SQ;
endrewards
```

Listing 5.1 shows a simplified module for a workflow W with two sequential tasks $\{T_1, T_2\}$, a deadline dw, a limited number of failures y, and a preference for cost reduction. The example

shows two actions: $[iSO]$ to execute task T_i using a small on-demand instance with 100% success probability or $[iSQ]$ to execute it in a small revocable instance with a 80% success probability. Each action has its expected duration (d_1 and d_2) and expected cost (e.g. c_{1SO}). The rewards are defined in terms of the costs, due to the user preference.

Before solving the MDP, it is verified that there exists at least one policy π that leads to a goal state $s \in G$. A model checking tool (PRISM (Kwiatkowska, Norman, & Parker 2011)) is used to verify this property. The same tool is used to construct the reachable state space, the transition matrix, and the reward matrix.

MDP Resolution: Classical approaches are used to solve the MDP constructed in the previous step. Classical approaches include value iteration, policy iteration, and linear programming. For a-cyclic MDPs, value iteration is guaranteed to terminate with optimal values in a single value update per state (Mausam & Kolobov 2012). Value iteration works by treating the value function as an assignment, starting with an initial value V^0 and successively approximating V^* with a V^m function such that the sequence of V^ms converges to V^* in the limit, as m tends to infinity. The value iteration algorithm is used to find the optimal policy π'^* for the reduced workflow w'.

5.3.3 Execution

This stage takes as input the optimal policy π'^* generated for the reduce workflow w' and translates it to the optimal policy π^* as described before. The online execution consists in taking the actions dictated by the policy $\pi^*(s)$ in every state s, starting from the initial state s_0 up to a goal state $s \in G$.

5.4 Case Study

As a case study, the planning approach is applied to the Epigenomics scientific workflow (Juve, Chervenak, Deelman, Bharathi, Mehta, & Vahi 2013), a well-known workflow that is extensively used in DNA research for mapping the epigenetic state of human cells. This case study serves not only to illustrate the use of the approach, but also to compare with other alternative solutions.

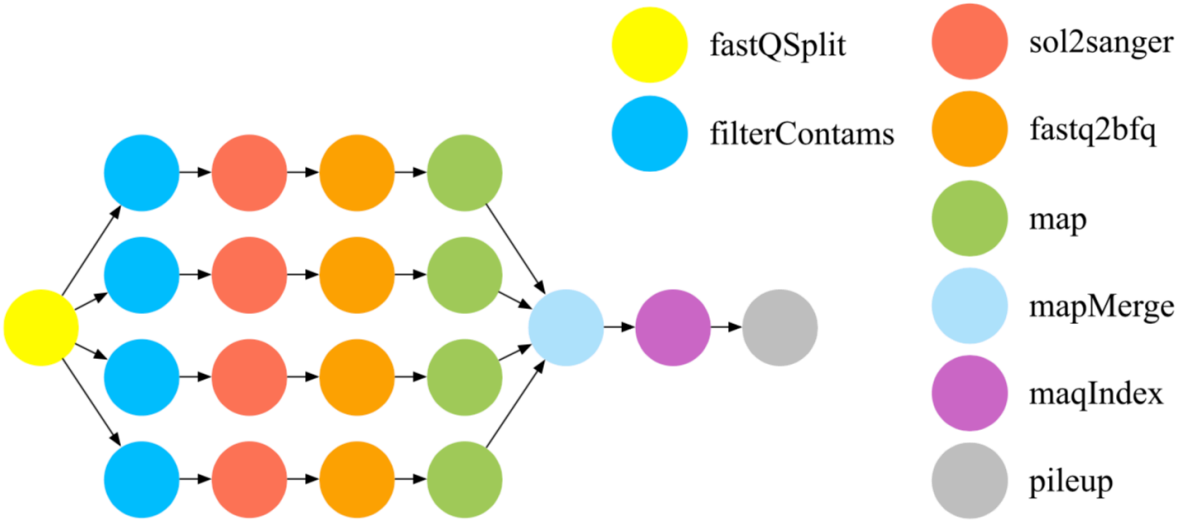

Figure 5.5: Case Study: Epigenomics workflow

Epigenomics is characterized as a highly pipelined workflow with multiple pipelines operating on independent chunks of data in parallel, as shown in Fig. 5.5. The input to the workflow is DNA sequence data obtained for multiple lanes from the genetic analysis process. The information from each lane is split into multiple chunks by the *fastQSplit* task. The number of splits generated depends on the partitioning factor used on the input data. The *filterContams* task then filter out noisy and contaminated data from each of the chunks. The data in each chunk are then converted to a format understood by the Maq DNA sequence mapping software by the *sol2sanger* task. For faster processing and reduced diskspace usage, the data is then converted to the binary fastQ format by *fastq2bfq*. Next, the remaining sequences are aligned with the reference genome by the *map* task. The results of individual map processes are combined using one or more stages of the *mapMerge* task. After merging, the *maqIndex* task operates on the merged alignment file and retrieves reads about a specific region. Finally, the *pileup* task reformats the data so that it can be displayed by a GUI. In most epigenomics jobs the CPU utilization is high and the application as CPU-bound (Juve, Chervenak, Deelman, Bharathi, Mehta, & Vahi 2013).

Let's consider an instance of the application defined by the workflow W. The workflow graph G_W, shown in Fig. 5.5, defines four concurrent pipelines for the task sequence $\langle$*filterContams*, *sol2sanger*, *fastq2bfq*, *map*$\rangle$, for a total number of tasks $n = 20$. The time limit for the execution of all tasks is $d_W = 71s$. Additionally, the user imposes a preference for minimizing monetary costs of the execution ($u = 1$).

To optimize the execution of a CPU-bound application, the system resorts to compute

Job	Count	Runtime		I/O read		I/O write		Peak memory		CPU utilization	
		Mean (s)	Std. dev.	Mean (MB)	Std. dev.	Mean (MB)	Std. dev.	Mean (MB)	Std. dev.	Mean (%)	Std. dev.
fast2bfq	128	1.40	0.24	10.09	1.73	2.22	0.40	4.05	0.01	88.42	0.10
pileup	1	55.95	0.00	151.82	0.00	83.95	0.00	148.26	0.00	153.43	0.00
mapMerge	8	11.01	12.18	27.68	32.49	26.71	32.89	5.00	0.39	95.08	0.06
map	128	201.89	21.91	138.76	0.80	0.90	0.22	196.04	5.50	96.69	0.01
sol2sanger	128	0.48	0.14	13.15	2.25	10.09	1.73	3.79	0.00	65.17	0.12
filterContams	128	2.47	0.43	13.25	2.26	13.25	2.26	2.97	0.06	88.54	0.09
mapIndex	1	43.57	0.00	214.10	0.00	107.53	0.00	6.17	0.00	99.50	0.00
fastQSplit	7	34.32	8.94	242.29	82.60	242.29	82.60	2.80	0.00	22.41	0.03

Figure 5.6: Case Study: Task characterization (c5.2xlarge machine)

optimized *c5* machines from Amazon Ec2. The set of available machines to the application is composed by four different sizes: *c5.2xlarge* (8 CPU), *c5.4xlarge* (16 CPU), *c5.9xlarge* (36 CPU) and *c5.18xlarge* (72 CPU). The notation $n\times$ is used: e.g., $2\times$ identifies a *c5.2xlarge*. Machines can be rented as on-demand O or revocable Q instances. The price for on-demand instances grows linearly with the number of CPUs starting in $c_{2x,O}(t) = 0.34\$/h = 0.0057\$/s$. The expected paid price for a revocable instance behaves as $c_{2x,Q}(t) = \beta \times c_{2x,O}(t)$, with $\beta = 0.2$ when the bid equals the on-demand price. Additionally, the probability of revocation remains fairly constant for $t < 120s$, such that $p_{2x,Q}(t) \approx 0.8$.

As described in the previous section, this approach requires a preparation stage that consists of 3 steps, namely: task characterization, workflow graph reduction, machine pre-selection.

Task Characterization: Task characterization consists in learning the behaviour of each task T when executed in different machines $v \in V$. Fig. 5.5 shows the characterization of the tasks that compose the workflow when executed on a *c5.2xlarge* machine. The performance model assumes that all tasks benefit from multicore processing, such that the runtime is inversely proportional to the number of cores available to their execution. The expected duration of each task is estimated in time slot of 1 second (e.g. $d_{T1,2xO} = 1$). The cost is defined w.r.t. the cheapest instances ($c_{2x,O}(1s) = 0.0057\$$), such that $c(T, v) = 100 \times c_v(d(T, v))/0.0057\$$.

Workflow Graph Reduction: The workflow graph G_W is composed of $n = 20$ nodes, including a set of $z = 4$ concurrent pipelines. The reduced workflow graph $G_{W'}$ consists $n' = 8$ sequential nodes $\{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8\}$. $G_{W'}$ identifies four task in the pipelines; in particular: T_2 for *filterContams*, T_3 for *sol2sanger*, T_4 for *fastq2bfq*, and T_5 for *map*. The cost of executing a tasks $T \in \{T_2, T_3, T_4, T_5\}$ in all pipelines is $4 \times c(T, v)$, for any machine v.

Machine Pre-Selection: Given the workflow W and the task characterization, a subset of machines is re-selected for the execution of each task, that defines the set V'_W. Table 5.1 presents the selected machine types (identified by their size) for each tasks $T_i \in G_{W'}$ with $d_W = 71s$. The original set $V_{W'}$ is composed of 64 choices of machines types $v \in V$. The machine pre-selection results in a set $V'_{W'}$ composed by 36 choices (counting on-demand and revocable instances).

Table 5.1: Preparation: Machine Pre-Selection

T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**
4x	9x	2x	18x	18x	18x	18x	18x
2x	4x		9x		9x	9x	9x
	2x		4x		4x		
					2x		

5.5 Evaluation

This approach has been evaluated in terms of parameter sensitivity and planning algorithms. The solution is compared to previous heuristics for the epigenomics application:

5.5.1 Quality and State Space

The sensitivity of the approach to the parameters that affect the search space has been evaluated. These parameters are: the number of failures y, the set of machines V and the deadline d_W. The *slack time* is defined as the time distance between the deadline and the minimum makespan: $ST = d_W - M(W, V_W)$. It is presented as a percentage of the minimum makespan as: $ST\% = (d_W - M(W, V_W))/M(W, V_w) \times 100$, and vary it between 0% and 100%.

Sensitivity to V': The subset of machines types limits the number of possible execution actions for every task in the workflow and the reachable state space. It is important to assess the impact of the composition of this set on the quality of the solution. Four sets of machine types are considered:

- V, the combination of all machines types for each task.
- V', defined by the machine pre-selection procedure.

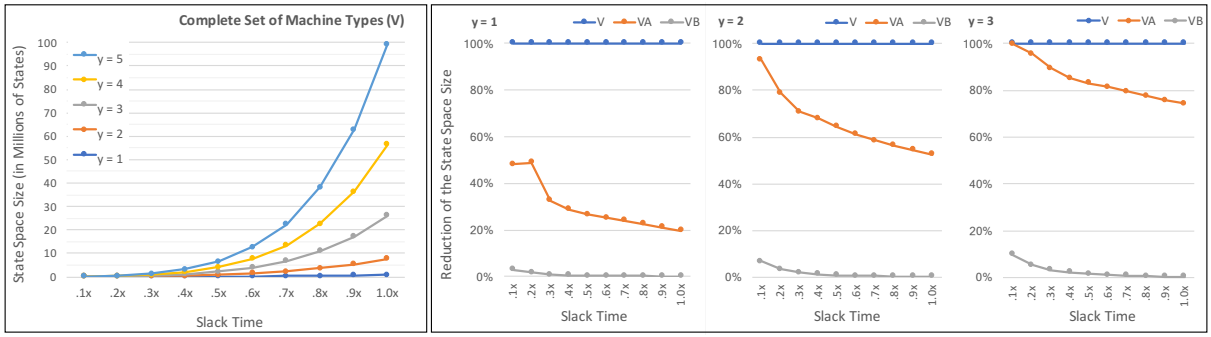

Figure 5.7: Scalability: Size of the reachable state space for V. Reduction of the reachable state space for V', V'_A and V'_B as percentage of the entire space ($y = 1, 2, 3$).

- V'_A, includes the fastest machines on-demand and all revocable machines, out of V'.
- V'_B, includes the fastest machines on-demand and fastest revocable machines, out of V'.

Figure 5.7 shows the size of the reachable state space for the entire set of machines V. The state space increases rapidly as a function of the slack time and the number of failures (for V and $y = 5$, there are 100 million reachable states). Fig. 5.7 shows the size of the state space for V', V'_A and V'_B w.r.t to the original size using V, for $y = 1, 2, 3$ ($y \geq 4$ behaves similar to $y = 3$). As expected, the set V' has no effect on the number of reachable states, as the machine pre-selection procedure is intended to reduce the number of action choices, instead. V' reduces the number of action choices by 16.2%, in comparison to V. Differently, the machine sets V'_A and V'_B have a significant impact on the size of the reachable state space. On average, the state space using V'_A represents between 29% and 85% of the original size, while the state space using V'_B represents 0.7% to 2.5% of the original size.

More importantly, the increase in the cost of the solution due to a reduced set of machines is small, for both sets V'_A and V'_B (less than 5% in the experiments). On average, $C_{V'_A}(\pi*) = 1.0142 \times C_V(\pi*)$ and $C_{V'_B}(\pi*) = 1.0429 \times C_V(\pi*)$. The set V' has no impact in solution costs.

Sensitivity to y: The maximum number of failures y limits the number of times a task can be executed using revocable instances and assumes values between 0 and d_W. To evaluate the impact of the the number of failures y in the quality of the solution, it has been computed the percentage of cost savings with respect to a baseline solution using only on-demand instances (equivalent to setting $y = 0$), for a series of slack time percentages from 0.1 to 1.0, using set V'_B. Results in Fig. 5.8 suggest that executing tasks in revocable machines yield significant

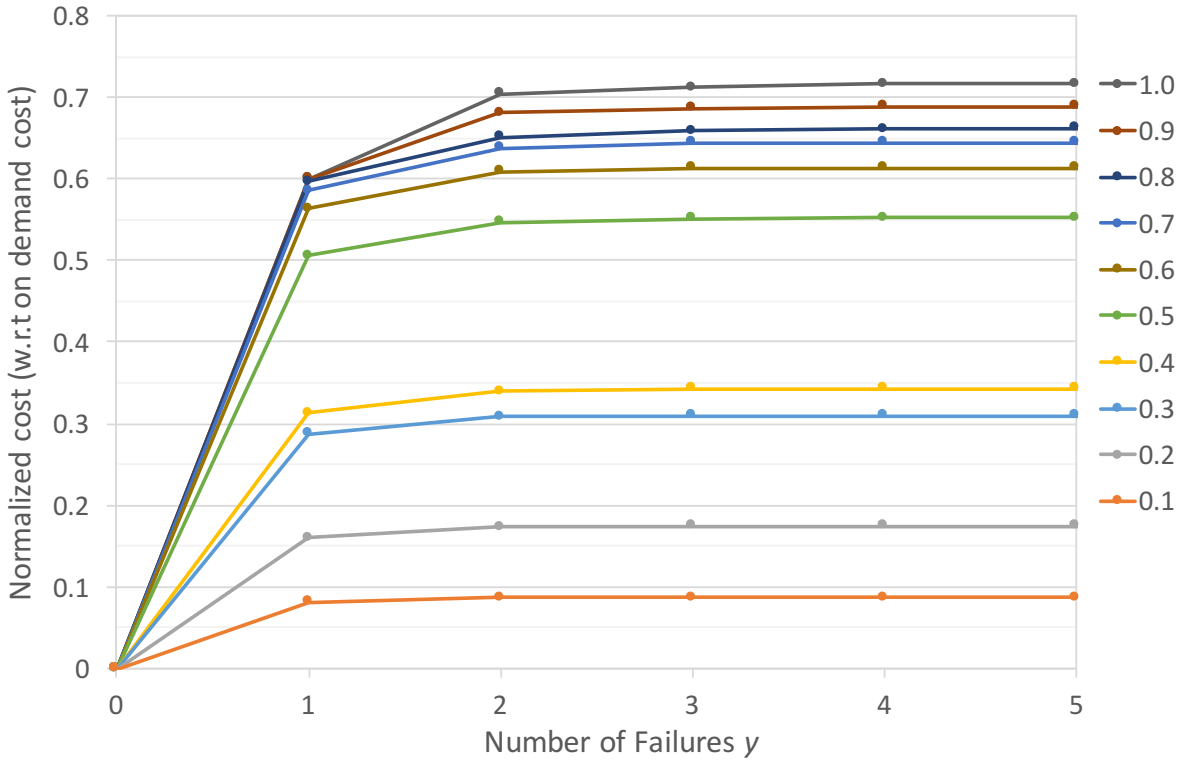

Figure 5.8: Scalability: Sensitivity to parameter y

cost reductions for any time slack time. The curves converge sharply, meaning that $y = 1, 2, 3$ produce the most significant impact in cost reductions and any solution derived using $y > 3$ yield little extra savings. Indeed, in the experiments, the improvement in cost savings from $y = 3$ to $y = 4$ is less than 0.01% for all slack times. In the rest of the experiments, $y = 3$ is used, since it provides a good trade-off between size of the state space and quality of the solution.

5.5.2 Planning Algorithms

Two techniques to solve the MPD have been tested: value iteration and path planning. Path planning differentiates from classical planning in that it performs best for problems with: (1) a low dispersion rate, i.e. from any state there are only a few reachable successive states; and (2) a clearly defined initial state and the goal states, such that the agent looks for a path between them. A path planning algorithm has been tested for comparison sake. This algorithm explored exhaustively the space of possible paths and was effective at finding the optimal solution; yet, it required longer search times.

As an example, this experiment was run with V'_B, $y = 1$, and $d_W = 71$. The space of reachable states consists of 440 states. Both techniques solve the MDP successfully. The value iteration algorithm finds the optimal policy and generates the optimal strategy in only 0.25 seconds. The path planning algorithm took 1153 seconds to find 82 policies that lead the execution to a goal state, for any possible execution history. Both algorithms generate solutions with the same utility.

5.5.3 Planning vs. Heuristics

Planning implemented with value iteration *VI-MDP* was compared to two previous solutions that implement heuristics: *LTO* and *Dyna*.

1. *LTO* (Poola, Ramamohanarao, & Buyya 2014): *LTO* is a running algorithm that executes each task using revocable machines of a single type (the cheapest machine type) multiple times until the task is executed successfully or until the latest time to on-demand (*LTO*); i.e. the minimum time required to execute all remaining tasks using on-demand instances. The *LTO* is recomputed every time a new task must be launched. Departing from an estimation of the execution time of every remaining task in every machine type, *LTO*'s conservative algorithm selects a combination of machine types that satisfies the deadline constraints and minimizes the cost (using the lowest cost machine types). Note that the *LTO* algorithm does not limit the number of failures and is equivalent to considering $y = d_W$

2. *Dyna* (Zhou, He, & Liu 2016): *Dyna* is an algorithm that exploits revocable machines to reduce costs while meeting probabilistic deadline guarantees. It searches the solution that minimizes costs in the on-demand domain, using the A^* algorithm. To the selection of a machine type for on-demand execution of a given tasks, Dyna adds new machine types for the execution in revocable instances, under two conditions: (1) the expected cost of the hybrid machines is less than the cost using only the on-demand instance (2) the deadline guarantees are not violated.

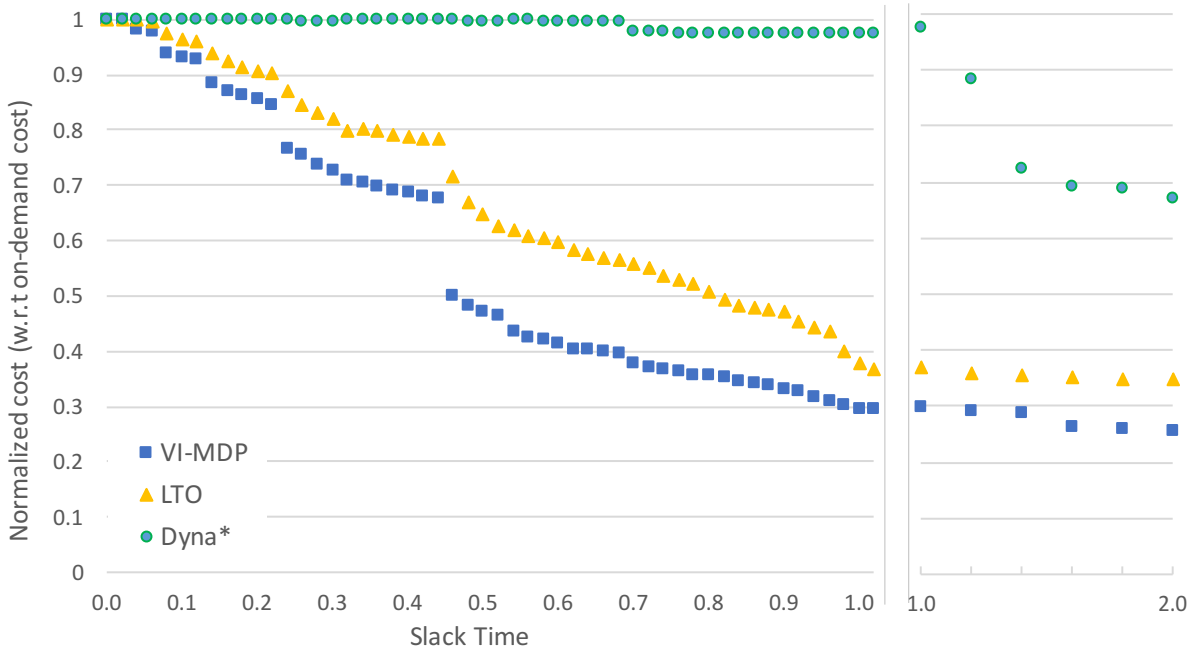

Figure 5.9: Cost reduction: Normalized cost (w.r.t on-demand cost) against the slack time

Comparison *VI-MPD* was simulated with V'_A and $y = 3$. Both *LTO* and *Dyna* are simulated with the machine set V' (see Table 5.1), with fixed task execution times, and deadline guarantees of 100% (note that V' is a superset of V'_A; *LTO* and *Dyna* have been given more freedom). For *LTO*, the solution considers both the conservative and aggressive algorithms. For *Dyna*, a variation (referred to as *Dyna**) was used. It selects the cheapest set of on-demand machine types using exhaustive exploration, instead of the A^* algorithms. Both *LTO* and *Dyna** resolve a pre-selection of machine types different from that of *VI-MPD*, such that the selected set leaves a smaller spare time for the execution of tasks in revocable instances.

Figure 5.9 shows the expected costs for the best solution of *VI-MDP*, *LTO*, and *Dyna**, as a function of the available slack. The cost is normalized with regard to the cost of running the workflow using just on on-demand instances (i.e., the worst case). Note that when there is no slack, all solutions have the same performance, because only the use of on-demand resources satisfy the deadline. As the slack increases, there are more opportunities to use revocable instances. Naturally, when the slack is very large (right side of the figure), it becomes possible to tolerate a large number of evictions and the problem becomes much simpler; in this case, any of the three algorithms can bring substantial advantages. Clearly, more powerful algorithms are needed when the slack time is relatively small and, since the algorithm has few attempts to execute tasks in revocable instances, these must be carefully selected. In this case, *Dyna**

performs very poorly; This can be explained by the algorithm, that fist selects a set of machines on-demand that satisfy the deadline and only after it adds on-demand instances for the spare time. *LTO* performs closer to *VI-MDP*, but *VI-MDP* still offers significant additional savings, from 20% to 38% depending on the slack time available for the workflow execution.

5.5.4 Discussion

The proposed approach resorts to a definition of the problem as a finite-horizon Markov Decision Process with rewards. System requirement regarding machine sizes for task execution and the structure of the workflow graph are encoded during the preparation and construction of the MDP. Preferences over the system performance and operational costs are encoded in the definition of the reward function. In particular, monetary cost of resources are encoded as cost in the execution of actions in the MDP, while restrictions on the execution times (deadlines) are encoded as high penalties to any state where the workflow execution fails to finish before the deadline. The solution of the MDP ensures that deadlines are met and monetary costs are minimized.

In terms of responsiveness, the execution plans generated offline already account for the occurrence of task failures due to instance revocations. That is, the policy contains contingency plans to react to failures as they occur at run-time. Thus, alternate plans can be enacted immediately, without the need for further planning.

The MDP solver implements the value iteration algorithm. By construction, the MDP generates all valid states of the system (consider the reduces workflow graph), and value iteration explores all the possible combinations to find the optimal policy. The search times are rather small, in the order of minutes. However, since the proposed approach is supposed to be executed offline, it is not affected by the longer search times. The major limitation in terms of scalability has to do with computational resources, since the controller must be able to store a large MDP and resolve it locally (more advanced implementations of the algorithm could also exploit parallel processing).

5.6 Related Work

The deployment of workflow applications in cloud environment has been studied in the past, mostly via heuristics that provision and allocate virtual machines on-demand in a sub-optimal manner (Kllapi, Sitaridi, Tsangaris, & Ioannidis 2011; Mao & Humphrey 2011; Byun, Kee, Kim, & Maeng 2011; Zhang, Cao, Hwang, & Wu 2011; Maguluri, Srikant, & Ying 2012). While the use of heuristics is justified for real-time applications, not all workflows require online scheduling. Differently, the approach presented in this chapter exploits the benefits of planning to generate the optimal solution offline and execute it at run-time. Also, differently from previous approaches, the construction of a complete MDP and offline planning allows to search the space of all virtual machine types available that can be candidates to the execution of tasks in the workflow.

The approach presented in this chapter exploits the opportunities offered by revocable instances via planning and is built on top of studies that suggest that cloud applications could resort to trivial bidding strategies and still achieve optimal cost and availability (Sharma, Irwin, & Shenoy 2016). Other proposals have attempted to deploy workflow applications using revocable instances, however, using heuristic algorithms. In (Poola, Ramamohanarao, & Buyya 2014) it is presented a heuristics that executes each task using the cheapest revocable machines, multiple times until the task is executed successfully or until the latest time to on-demand (LTO); i.e. the minimum time required to execute all remaining tasks using on-demand instances. The LTO is recomputed every time a new task must be launched, using the cheapest (or most expensive) combination of on-demand machine types. Greedy heuristics like this perform poorly under strict deadlines that allow for little slack time, since the algorithms cannot plan ahead and foresee the use of slack time for the execution of more expensive tasks. Alternatively, Dyna (Zhou, He, & Liu 2016) is a solution that approximates the benefits of planning. However, it does not explore the space of on-demand and revocable instances together. Instead, it searches a cheap solution in the on-demand domain and further optimizes the solution by adding revocable instances to each task, only if beneficial to reduce the task cost without violating the probabilistic deadline guarantees. Therefore, it may discard relevant solutions, including the optimal one, and fail at exploiting revocable instances with an overall view of all tasks in the workflow.

5.7 Conclusions

The approach presented above aims at responding the question: How can probabilistic and temporal planning be used for the (offline) generation of policies for the execution of workflows using spot instances in cloud environments?.

The proposed approach solves a probabilistic planning problem using a Markov Decision Process model and value iteration algorithms. This approach was successful at meeting the goals and addressing the limitation of previous work in that:

1. Using a probabilistic model that captures task execution costs and durations in on-demand and spot instances, planning can produce schedules that foresee the selection of spot instances for the execution of the most expensive tasks in the workflow, minimizing the overall expected cost. This improves in previous cost-aware techniques that employ on-demand instances only or that use heuristics that cannot lead to the minimum cost solution.

2. The generated policies can account for the possibility that task executions fail due to spot instance revocations, and have alternative execution schedules to guarantee that the workflow execution meet its deadline with 100% probability. It improves on previous work that can only produce schedules that give probabilistic guarantees.

Summary

This solution demonstrated that planning the execution of workflows in the cloud can benefit from searching the space of virtual machines types and mixed on-demand and revocable instances, to reduce execution costs while meeting deadline constraints. The planning approach is implemented departing from a finite horizon Markov Decision Process, later solved with a value iteration algorithm. Techniques to reduce the state space without loosing value of the solution were also presented. The results of applying this solution to a real scientific workflow suggest that it can achieve significant cost savings with respect to previous solutions that implement heuristics, when the slack time allows to use revocable machines.

Publications

The work presented in this chapter has contributed to the following publication:

Planning Workflow Executions when Using Spot Instances in the Cloud. R. Gil Martinez, A. Lopes, L. Rodrigues. Proceedings of the ACM Symposium on Applied Computing (SAC). Limassol, Cyprus. 2019

6 Conclusions and Future Work

This chapter closes the thesis. Section 6.1 summarizes the main results presented in the thesis and briefly presents additional results that emerged from the work previously presented. It closes by providing an answer to the questions introduced in Chapter 1. Section 6.2 concludes the thesis by discussing future research directions.

6.1 Conclusions

The thesis has proposed, developed, and evaluated solutions based on three distinct scenarios for the use of automated planning to support the deployment and management of computer applications running in cloud environments. The benefits of each of these contributions were illustrated through case studies. The considered solutions can be grouped in two families:

- solutions that use planning to support elastic scaling of resources for cloud applications. This family of solutions uses a planner to directly manipulate the selection of machines sizes that compose a pool of servers, used by a user-interactive application to provide its services; and
- solutions that use planning to support the deployment of workflow applications in cloud environments. This family of solutions profits from dynamic provisioning of resources to deploy tasks in a workflow, scheduled according to a policy defined by the planner.

For the family of solutions that uses planning to support elastic scaling of resources, two distinct approaches where studied:

The *(offline) generation of reactive policies*, presented in Chapter 3. This solution relies on AI planning languages and tools to generate policies that support elastic scaling. It is effective at mapping a (simplified) architectural model of the system into a standard action language (PDDL3.0), at scanning the space of system conditions, at generating a set of plans produced by off-the-shelve temporal planners (TFD), and at selecting the best configurations and plans

according to multi-objective criteria. The results suggest that automated planning is a valid alternative for the offline generation of policies applicable to real life cloud environment scenarios. In particular, planning can circumvent the limitations of human-defined policies, producing more elaborate combinations of actions that lead the system to the desired configuration, while preventing the violation of system constraints. Yet, AI planning has a long way to go in terms of language support by the tools and efficiency, as long search times prevent these solutions to be used for online planning.

The *(online) generation of proactive plans*, presented in Chapter 4. The solution introduced Augure, a proactive decision-making tool for resource adaptation in cloud-enabled applications. Augure combines the benefits of long-term predictions of the workload curve and the use of heterogeneous resources to find adaptation plans that offer a good fitting of resources to the demand in the long horizon. Augure plans minimize the price billed by the cloud provider and mitigate the impact of reconfiguration on the system performance. Using off-the-shelf solvers, Augure can search the space of resource combinations and action schedules in less than 5 minutes. Augure recognizes workload behavior changes and adjusts the prediction promptly. Augure is able to tame the number of reconfigurations when compared to greedy proactive techniques (such as Vadara+), achieving better results and lower costs overall.

Combining Approaches

Although the thesis has introduced the idea that combining offline and online planning techniques can be done, the problem of combining these two solutions was not explicitly addressed. To control the resources that compose the pool of servers in a cloud application, the offline generation of policies can provide the manager with a preliminary set of plans that can be enacted under known conditions. A proactive controller would execute these plans whenever the environment is expected to stay steady in the horizon and the system conditions match those defined in the reactive policy. On the other hand, the plans generated online by the proactive controller can be stored and classified, such as to define reactive policies under common conditions, both in terms of system initial state and the predicted behavior of the workload curve. Indeed, deliberate planning (online) and reactive policies (offline) operate at two complementary levels of control, that is consistent with the definition of goal management and change management, from the perspective of layered control in architecture-based self-adaptation.

For the family of solutions that uses planning to support the deployment of workflow applications in cloud environments, one solution was studied:

The *(offline) generation of execution policies*, presented in Chapter 5. This solutions demonstrated that planning the execution of workflows in the cloud can benefit from searching the space of virtual machines types and mixed on-demand and revocable instances, to reduce execution costs while meeting deadline constraints. The planning approach is implemented departing from a finite horizon Markov Decision Process, later solved with a value iteration algorithm. Techniques to reduce the state space without loosing value of the solution were also presented. The results of applying this solution to a real scientific workflow suggest that it can achieve cost savings from 20% to 38% with respect to previous solutions that implement heuristics. The solution saves up to 73% with respect to on-demand solutions. All experiments consider that deadlines are not tight, such that planning can take advantage of slack time to attempt task executions using revocable machines.

Overall, three different planning techniques have been successfully implemented to support the deployment and management of applications in cloud environments: AI Planning, Linear Programming, and Markov Decision Processes. For comparison purposes, AI Planning techniques has been tested in all three scenarios; whereas it is partially possible to express the complexity of the planning problems of all three adaptation scenarios using standard planning languages, off-the-shelf planners were not efficient enough at exploring the solution space and finding optimal plans in short times. Ideally, one would like to combine the expressiveness of standard AI planning languages (e.g. PDDL) with the efficiency of mature search algorithms and tools (e.g. constraint solvers). The efficiency of the tools is, therefore, the last gap to be bridged for the full integration of automated planning (specifically, AI planning) into the self-adaptation of systems.

Limitations

The use of planning for the purposes of self-adaptation in cloud environments still presents important limitations. Ideally, decision-making in self-adaptive systems (planning, in particular) is supported by knowledge about the system and its environment. Knowledge regarding the system architecture and constraints, the adaptation actions and their effects, and the business goals and preferences is presumed to be accurate. However, building a realistic model that

captures with precision all parameters and external factors that potentially affect decision-making is practically impossible. On top of that, planning languages may be limited in the description of the world that can be captured. Even more troublesome, AI planning tools are limited in their support of these languages and somewhat primitive in their ability to prune the search space and resolve highly complex problems in narrow times. It is expected that the evolution of AI planning will lead to more practical technologies that would widen the applicability of these techniques to real life systems.

Ramifications and Collaborations

Additionally to the work presented in previous chapters, the research in the topic of planning to support self-adaptation of cloud applications led to collaboration in other solutions.

In particular, the *(online) revision of action models*. Given that many systems are affected by uncertainty due to hardware heterogeneity, co-residency and failures, adaptation actions often have non-deterministic impacts, potentially leading to multiple outcomes. When this uncertainty is not captured explicitly in the models that guide adaptation, decisions may turn out ineffective or even harmful to the system. Also critical is the need for these models to be readable to human operators accountable for the system. This study proposed a method to learn human-readable models that capture non-deterministic impacts explicitly, using the K-plane clustering technique in a novel way. Additionally, expert's knowledge is exploited to bootstrap the adaptation process as well as how to use the learned impacts to revise models defined offline.

The work of this collaboration has been published through the following publication:

Learning Non-Deterministic Impact Models for Adaptation. F. Duarte, R. Gil Martinez, P. Romano, A. Lopes, L. Rodrigues. Proceedings of the 13th IEEE/ACM International Symposium on Software Engineering and Self-Managing Systems (SEAMS@ICSE). Gothenburg, Sweden. 2018

6.2 Future Work

As discussed previously, additional research efforts are necessary to fully understand the strengths and limitations that arise from the combination of the use of online learning and au-

tomated planning to support the management of resources in the cloud; in particular, elastic scaling. In the following, a possible research vector that can be pursued considering the contributions of thesis, and the combination of the discussed approaches, is motivated and briefly presented.

From the perspective of architecture-based self-adaptation with layered control, it is desired an external controller that can operate at a low-level of corrective control (via guarded actions triggered by undesired events) and high-level deliberate planning (via a set of decision steps that lead the system to its optimal state). These operational instructions are expected to manage the system under uncertainty. This uncertainty does not only make so that adaptation action have multiple possible outcomes (as resolved in collaboration work presented above). Uncertainty can also affect the prediction of the behavior of the environment and the evolution of the system guided by a deliberate plan. In an integrated solution, it is important to define mechanisms that combine effectively the corrective control and deliberate planning, whenever the system behavior deviates from the behavior expected by the model. The interplay between these two layers is still to be explored fully. Additionally, the integration of learning techniques to assess the effectiveness of reactive actions and plans at run-time, must be studied. Full self-adaptation, after all, can only be achieved when the system is able to learn from the run-time execution of pre-computed plans and policies.

In addition, from a software engineering perspective, the definition of an Architecture Description Language (ADL) that is specific to the deployment and management of applications in the cloud infrastructure could be beneficial. In particular, engineers could more easily express knowledge about the system and its adaptation using such ADL, which may be automatically translated into a standardized planning language (e.g. PDDL). Studying such integration is left for future work.

Bibliography

Alipourfard, O., H. H. Liu, J. Chen, S. Venkataraman, M. Yu, & M. Zhang (2017, March). Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017*, Boston, MA, USA, pp. 469–482.

Arlitt, M. & T. Jin (2000). A workload characterization study of the 1998 world cup web site. *IEEE network 14*(3), 30–37.

Baptiste, P., C. Le Pape, & W. Nuijten (2012). *Constraint-based scheduling: applying constraint programming to scheduling problems*, Volume 39. Springer Science & Business Media.

Barnes, J. M., A. Pandey, & D. Garlan (2013, November). Automated planning for software architecture evolution. In *IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*, Silicon Valley, CA, USA, pp. 213–223.

Benton, J., A. J. Coles, & A. Coles (2012, June). Temporal planning with preferences and time-dependent continuous costs. In *Proc. of International Conference on Automated Planning and Scheduling, ICAPS 2012*, Atibaia, São Paulo, Brazil.

Braberman, V. A., N. D'Ippolito, J. Kramer, D. Sykes, & S. Uchitel (2015, August). MORPH: a reference architecture for configuration and behaviour self-adaptation. In *Proc. of International Workshop on Control Theory for Software Engineering, CTSE@SIGSOFT FSE 2015*, Bergamo, Italy, pp. 9–16.

Brun, Y., C. Serugendo, Giovannaand Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, & M. Shaw (2009). Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*, pp. 48–70. Springer.

Byun, E., Y. Kee, J. Kim, & S. Maeng (2011). Cost optimized provisioning of elastic resources for application workflows. *Future Generation Computer Systems 27*(8), 1011–1026.

Cámara, J., D. Garlan, B. R. Schmerl, & A. Pandey (2015, April). Optimal planning for

architecture-based self-adaptation via model checking of stochastic games. In *Proc. of ACM Symposium on Applied Computing, SAC 2015*, Salamanca, Spain, pp. 428–435.

Chen, B., X. Peng, Y. Yu, B. Nuseibeh, & W. Zhao (2014, May). Self-adaptation through incremental generative model transformations at runtime. In *Proc. of International Conference on Software Engineering, ICSE 2014*, Hyderabad, India, pp. 676–687.

Cheng, S.-W. & D. Garlan (2012). Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software 85*(12), 2860–2875.

Cheng, S.-W., D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, & P. Steenkiste (2002). Using architectural style as a basis for system self-repair. In *Software Architecture: System Design, Development and Maintenance*, pp. 45–59. Springer.

Coker, Z., D. Garlan, & C. Le Goues (2015, May). SASS: self-adaptation using stochastic search. In *IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*, Florence, Italy, pp. 168–174.

Cooper, B., A. Silberstein, E. Tam, R. Ramakrishnan, & R. Sears (2010). Benchmarking cloud serving systems with YCSB. In *Proc. of ACM Symposium on Cloud Computing, (SOCC) 2010*, Indianapolis, USA, pp. 143–154.

da Silva, C. E. & R. de Lemos (2011). A framework for automatic generation of processes for self-adaptive software systems. *Informatica: An International Journal of Computing and Informatics 35*(1), 3–13.

Dantzig, G. (2016). *Linear programming and extensions*. Princeton university press.

Dashofy, E. M., A. van der Hoek, & R. N. Taylor (2002, November). Towards architecture-based self-healing systems. In *Proc. of Workshop on Self-Healing Systems, WOSS 2002*, Charleston, South Carolina, USA, pp. 21–26.

Delimitrou, C. & C. Kozyrakis (2013, March). Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013*, Houston, TX, USA, pp. 77–88.

Didona, D., P. Romano, S. Peluso, & F. Quaglia (2014). Transactional auto scaler: Elastic scaling of replicated in-memory transactional data grids. *ACM Transactions on Autonomous and Adaptive Systems, TAAS 2014 9*(2), 11:1–11:32.

Dobson, S., S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, & F. Zambonelli (2006). A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems, (TAAS) 2006 1*(2), 223–259.

Duan, R., R. Prodan, & T. Fahringer (2007, November). Performance and cost optimization for multiple large-scale grid workflow applications. In *Proc. of ACM/IEEE Conference on High Performance Networking and Computing, SC 2007*, Reno, Nevada, USA, pp. 12.

Elmore, A., V. Arora, R. Taft, A. Pavlo, D. Agrawal, & A. El Abbadi (2015). Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proc. of ACM International Conference on Management of Data (SIGMOD)*, Melbourne, Australia, pp. 299–313.

Eyerich, P., R. Mattmüller, & G. Röger (2009, September). Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proc. of International Conference on Automated Planning and Scheduling, ICAPS 2009*, Thessaloniki, Greece.

Fox, M. & D. Long (2003). PDDL2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR) 20*, 61–124.

Franco, J. M., F. Correia, R. Barbosa, M. Z. Rela, B. R. Schmerl, & D. Garlan (2016). Improving self-adaptation planning through software architecture-based stochastic modeling. *Journal of Systems and Software 115*, 42–60.

Frömmgen, A., R. Rehner, M. Lehn, & A. P. Buchmann (2015, July). Fossa: Learning ECA rules for adaptive distributed systems. In *IEEE International Conference on Autonomic Computing (ICAC)*, Grenoble, France, pp. 207–210.

Galante, G. & L. D. Bona (2012). A survey on cloud computing elasticity. In *Proc. of IEEE International Conference on Utility and Cloud Computing (UCC)*, Chicago, USA, pp. 263–270.

Ganek, A. G. & T. A. Corbi (2003). The dawning of the autonomic computing era. *IBM Systems Journal 42*(1), 5–18.

Garlan, D., S. Cheng, A. Huang, B. R. Schmerl, & P. Steenkiste (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer 37*(10), 46–54.

Gartner (2018). Gartner forecasts worldwide public cloud revenue to grow 21.4 percent in

2018. https://www.gartner.com/newsroom/id/3871416.

Gat, E. & R. P. Bonnasso (1998). On three-layer architectures. *Artificial intelligence and mobile robots 195*, 210.

Gerevini, A. & D. Long (2005). Plan constraints and preferences in pddl3. *The Language of the 5th International Planning Competition. Technical Report, University of Brescia, Italy 75*.

Gerevini, A., A. Saetti, & I. Serina (2006). An approach to temporal planning and scheduling in domains with predictable exogenous events. *Journal of Artificial Intelligence Research (JAIR) 25*, 187–231.

Ghezzi, C., L. S. Pinto, P. Spoletini, & G. Tamburrelli (2013, May). Managing non-functional uncertainty via model-driven adaptivity. In *International Conference on Software Engineering, ICSE 2013*, San Francisco, CA, USA, pp. 33–42. IEEE.

Gong, Z., X. Gu, & J. Wilkes (2010). PRESS: predictive elastic resource scaling for cloud systems. In *Proc. of International Conference on Network and Service Management (CNSM)*, San Francisco, USA, pp. 9–16.

Herbst, N., N. Huber, S. Kounev, & E. Amrehn (2014). Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and Computation: Practice and Experience 26*(12), 2053–2078.

Horn, P. (2001). Autonomic computing: Ibm's perspective on the state of information technology.

Hummaida, A. R., N. W. Paton, & R. Sakellariou (2016). Adaptation in cloud resource configuration: a survey. *J. Cloud Computing 5*, 7.

IBM (2016). CPLEX optimization studio.

Jacobson, D., D. Yuan, & N. Joshi (2013). Scryer: Netflix's predictive auto scaling engine. https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270.

Jiang, J., J. Lu, G. Zhang, & G. Long (2013). Optimal cloud resource auto-scaling for web applications. In *Proc. of IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Delft, Netherlands, pp. 58–65.

Jung, G., M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, & C. Pu (2010, June). Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *International Conference on Distributed Computing Systems, ICDCS 2010*, Genova, Italy, pp. 62–73.

Juve, G., A. L. Chervenak, E. Deelman, S. Bharathi, G. Mehta, & K. Vahi (2013). Characterizing and profiling scientific workflows. *Future Generation Computer Systems 29*(3), 682–692.

Kephart, J. O. & D. M. Chess (2003). The vision of autonomic computing. *Computer 36*(1), 41–50.

Kim, I., W. Wang, Y. Qi, & M. Humphrey (2016, June). Empirical evaluation of workload forecasting techniques for predictive cloud resource scaling. In *Proc. of IEEE International Conference on Cloud Computing, (CLOUD) 2016*, San Francisco, USA, pp. 1–10.

Kllapi, H., E. Sitaridi, M. M. Tsangaris, & Y. E. Ioannidis (2011, June). Schedule optimization for data processing flows on the cloud. In *Proc. of ACM International Conference on Management of Data, SIGMOD 2011*, Athens, Greece, pp. 289–300.

Kramer, J. & J. Magee (2007). Self-managed systems: an architectural challenge. In *Future of Software Engineering, 2007, FOSE 2007*, pp. 259–268. IEEE.

Krupitzer, C., F. M. Roth, S. VanSyckel, G. Schiele, & C. Becker (2015). A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing 17*, 184–206.

Kwiatkowska, M. Z., G. Norman, & D. Parker (2011, July). PRISM 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification, CAV 2011*, Snowbird, UT, USA, pp. 585–591.

Leitner, P. & J. Cito (2016). Patterns in the chaos - A study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology 16*(3), 15:1–15:23.

Lim, H., S. Babu, & J. S. Chase (2010, June). Automated control for elastic storage. In *Proc. of International Conference on Autonomic Computing, ICAC 2010*, Washington, DC, USA, pp. 1–10.

Liu, C., C. Liu, Y. Shang, S. Chen, B. Cheng, & J. Chen (2017). An adaptive prediction approach based on workload pattern discrimination in the cloud. *J. Network and Computer*

Applications 80, 35–44.

Loff, J. & J. Garcia (2014). Vadara: Predictive elasticity for cloud applications. In *Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Singapore, pp. 541–546.

Lorido-Botran, T., J. Miguel-Alonso, & J. Lozano (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing 12*(4), 559–592.

Ma, A., C. Zhang, B. Zhang, & X. Zhang (2016). Cost optimization oriented dynamic resource allocation for service-based system in the cloud environment. In *Proc. of IEEE International Conference on Web Services, ICWS 2016*, San Francisco, USA, pp. 700–703.

Maguluri, S. T., R. Srikant, & L. Ying (2012, March). Stochastic models of load balancing and scheduling in cloud computing clusters. In *Proc. of IEEE International Conference on Computer Communications, INFOCOM 2012*, Orlando, FL, USA, pp. 702–710.

Mao, M. & M. Humphrey (2011, November). Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011*, Seattle, WA, USA, pp. 49:1–49:12.

Marler, R. T. & J. S. Arora (2010). The weighted sum method for multi-objective optimization: new insights. *Structural and multidisciplinary optimization 41*(6), 853–862.

Mausam & A. Kolobov (2012). *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool Publishers.

McDermott, D., M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, & D. Wilkins (1998). *PDDL: the planning domain definition language*.

Méhus, J., T. V. Batista, & J. Buisson (2012). ACME vs PDDL: support for dynamic reconfiguration of software architectures. *Computing Research Repository (CoRR) abs/1206.0122*.

Meng, S., L. Liu, & V. Soundararajan (2010, November). Tide: achieving self-scaling in virtualized datacenter management middleware. In *Proc. of International Middleware Conference Industrial Track*, Bangalore, India, pp. 17–22.

Messias, V. R., J. C. Estrella, R. Ehlers, M. J. Santana, R. H. C. Santana, & S. Reiff-Marganiec (2016). Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. *Neural Computing and Applications 27*(8), 2383–2406.

Moreno, G. A., J. Cámara, D. Garlan, & B. R. Schmerl (2015, August). Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proc. of Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, Bergamo, Italy,, pp. 1–12.

Nazaruk, A. & M. Rauchman (2013). Big data in capital markets. In *Proc. of ACM International Conference on Management of Data, (SIGMOD) 2013*, New York, USA, pp. 917–918.

Neamtiu, I. (2011, May). Elastic executions from inelastic programs. In *Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2011*, Honolulu , HI, USA, pp. 178–183.

Nguyen, H., Z. Shen, X. Gu, S. Subbiah, & J. Wilkes (2013). AGILE: elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of International Conference on Autonomic Computing, (ICAC) 2013*, San Jose, USA, pp. 69–82.

Oreizy, P., D. Heimbigner, G. Johnson, M. M. Gorlick, R. N. Taylor, A. L. Wolf, N. Medvidovic, D. S. Rosenblum, & A. Quilici (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems 14*(3), 54–62.

Pandey, A., G. A. Moreno, J. Cámara, & D. Garlan (2016, September). Hybrid planning for decision making in self-adaptive systems. In *International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2016*, Augsburg, Germany, pp. 130–139.

Poola, D., K. Ramamohanarao, & R. Buyya (2014, June). Fault-tolerant workflow scheduling using spot instances on clouds. In *Proc. of International Conference on Computational Science, ICCS 2014*, Cairns, Queensland, Australia, pp. 523–533.

Qu, C., R. Calheiros, & R. Buyya (2016). Auto-scaling web applications in clouds: A taxonomy and survey. *Computing Research Repository (CoRR)*.

Qu, C., R. N. Calheiros, & R. Buyya (2018). Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR) 51*(4), 73.

Rajan, D., A. Canino, J. A. Izaguirre, & D. Thain (2011, November). Converting a high performance application to an elastic cloud application. In *International Conference on Cloud Computing Technology and Science, CloudCom 2011*, Athens, Greece, pp. 383–390.

Rasche, C. & S. Ziegert (2013). Multilevel planning for self-optimizing mechatronic systems.

In *5th ADAPTIVE*, pp. 8–13. Citeseer.

Rosa, L., L. Rodrigues, A. Lopes, M. A. Hiltunen, & R. D. Schlichting (2013). Self-management of adaptable component-based applications. *IEEE Transactions in Software Engineering 39*(3), 403–421.

Sakellariou, R., H. Zhao, E. Tsiakkouri, & M. D. Dikaiakos (2007). Scheduling workflows with budget constraints. In *Integrated Research in GRID Computing*, pp. 189–202. Springer.

Salehie, M. & L. Tahvildari (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 4*(2), 14.

Shariffdeen, R. S., D. T. S. P. Munasinghe, H. S. Bhathiya, U. K. J. U. Bandara, & H. M. N. D. Bandara (2016). Workload and resource aware proactive auto-scaler for paas cloud. In *Proc. of IEEE International Conference on Cloud Computing, (CLOUD) 2016*, San Francisco, USA, pp. 11–18.

Sharma, P., D. E. Irwin, & P. J. Shenoy (2016, June). How not to bid the cloud. In *USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016*, Denver, CO, USA.

Sharma, P., D. E. Irwin, & P. J. Shenoy (2017). Keep it simple: Bidding for servers in today's cloud platforms. *IEEE Internet Computing 21*(3), 88–92.

Shen, Z., S. Subbiah, X. Gu, & J. Wilkes (2011). CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proc. of ACM Symposium on Cloud Computing (SOCC)*, Cascais, Portugal, pp. 5.

Srirama, S. N. & A. Ostovar (2014). Optimal resource provisioning for scaling enterprise applications on the cloud. In *Proc. of IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2014*, Singapore, pp. 262–271.

Sterritt, R. (2003). Autonomic computing: the natural fusion of soft computing and hard computing. In *International Conference on Systems, Man and Cybernetics*, Volume 5, pp. 4754–4759. IEEE.

Sykes, D., W. Heaven, J. Magee, & J. Kramer (2007). Plan-directed architectural change for autonomous systems. In *Proc. of Conference on Specification and verification of component-based systems: Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT FSE*, pp. 15–21. ACM.

Taft, R., E. Mansour, M. Serafini, J. Duggan, A. Elmore, A. Aboulnaga, A. Pavlo, & M. Stonebraker (2014). E-store: Fine-grained elastic partitioning for distributed transaction processing. *Proc. of VLDB Endowment PVLDB 8*(3), 245–256.

Tajalli, H., J. Garcia, G. Edwards, & N. Medvidovic (2010, September). PLASMA: a plan-based layered architecture for software model-driven adaptation. In *IEEE/ACM International Conference on Automated Software Engineering, ASE 2010*, Antwerp, Belgium, pp. 467–476.

The Internet Traffic Archive (2018). 1998 world cup web site access logs. `http://ita.ee.lbl.gov/traces/WorldCup/WorldCup.html`.

Tichy, M. & B. Klöpper (2012). Planning self-adaption with graph transformations. In *Applications of Graph Transformations with Industrial Relevance*, pp. 137–152. Springer.

Trushkowsky, B., P. Bodík, A. Fox, M. Franklin, M. Jordan, & D. Patterson (2011). The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proc. of USENIX Conference on File and Storage Technologies, (FAST) 2011*, pp. 12–12.

Urdaneta, G., G. Pierre, & M. van Steen (2009). Wikipedia workload analysis for decentralized hosting. *Computer Networks 53*(11), 1830–1845.

Vallati, M., L. Chrpa, M. Grzes, T. L. McCluskey, M. Roberts, & S. Sanner (2015). The 2014 international planning competition: Progress and trends. *AI Magazine 36*(3), 90–98.

Vallati, M., L. Chrpa, & T. L. McCluskey. The 2014 international planning competition: Description of participating planners.

Verma, M., G. Gangadharan, N. Narendra, V. Ravi, V. Inamdar, L. Ramachandran, R. Calheiros, & R. Buyya (2016). Dynamic resource demand prediction and allocation in multi-tenant service clouds. *Concurrency and Computation: Practice and Experience 28*(17), 4429–4442.

Wang, C., A. Gupta, & B. Urgaonkar (2016). Fine-grained resource scaling in a public cloud: A tenant's perspective. In *Proc. of IEEE International Conference on Cloud Computing, (CLOUD) 2016*, San Francisco, USA, pp. 124–131.

Weyns, D. (2019). Software engineering of self-adaptive systems. In *Handbook of Software Engineering.*, pp. 399–443.

Yu, J., R. Buyya, & C. Tham (2005, December). Cost-based scheduling of scientific workflow application on utility grids. In *Conference on e-Science and Grid Technologies, e-Science 2005*, Melbourne, Australia, pp. 140–147.

Zhang, F., J. Cao, K. Hwang, & C. Wu (2011, November). Ordinal optimized scheduling of scientific workflows in elastic compute clouds. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2011*, Athens, Greece, pp. 9–17.

Zheng, L., C. Joe-Wong, C. W. Tan, M. Chiang, & X. Wang (2015). How to bid the cloud. *Computer Communication Review 45*(5), 71–84.

Zhou, A. C., B. He, & C. Liu (2016). Monetary cost optimizations for hosting workflow-as-a-service in iaas clouds. *IEEE Transactions on Cloud Computing 4*(1), 34–48.

Ziegert, S. & H. Wehrheim (2015). Temporal plans for software architecture reconfiguration. *Computer Science - R&D 30*(3-4), 303–320.