Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Optimized Deployment of Multi-cloud Applications via HTN Planning

Faisal Khan

Course of Study:	Computer Science
Examiner:	Dr. Ilche Georgievski
Supervisor:	Dr. Ilche Georgievski
Commenced:	December 22, 2022
Completed:	June 23, 2023

Abstract

In this thesis, we present an end-to-end closed system that addresses the aforementioned challenges, taking into account the multi-cloud factor. Our solution leverages a combination of techniques, including Hierarchical Task Networks (HTN) planning, to optimize infrastructure across multiple cloud providers. By analyzing the current state of the infrastructure and utilizing time-series forecasting, we accurately predict future resource usage patterns. These insights are then fed into the HTN Planner (specifically, the GTPyhop Planner), enabling the generation of optimized plans that consider the multicloud environment. By executing the generated plans within the infrastructure, our solution reduces costs, optimizes resource allocation, and minimizes resource wastage across multiple cloud providers. We provide a comprehensive approach to address the challenges associated with cloud-based deployments, taking into account the multicloud factor and the intricacies of managing resources across different cloud providers. This research contributes to advancing cloud computing by providing a holistic solution that enhances resource utilization, mitigates latency issues, optimizes infrastructure, and manages resources effectively in multicloud environments using HTN planning. The proposed system enables organizations to optimize their infrastructure across multiple cloud providers, leading to improved operational efficiency, reduced costs, and enhanced performance in their cloud-based deployments.

Contents

List of Figures

List of Listings

1 Introduction

Cloud computing has transformed the way organizations manage their IT infrastructure, offering unparalleled scalability, flexibility, and on-demand resource provisioning. However, as cloud deployments have become more complex and widespread, organizations face challenges in managing costs and optimizing their infrastructure. To overcome these challenges, innovative approaches such as multi-cloud deployments have emerged as viable solutions.

Cloud cost reduction and infrastructure optimization are crucial goals for organizations seeking to maximize the benefits of cloud computing. The shift from capital expenditure (CapEx) [1] to operational expenditure (OpEx) [2] models has introduced new cost management challenges. While the pay-as-you-go model provides flexibility, it can also result in unanticipated costs. Factors such as underutilization of resources, inefficient workload placement, and lack of visibility into usage patterns contribute to cost optimization difficulties. Strategies aimed at optimizing cloud costs strive to ensure optimal performance and resource utilization while minimizing expenses.

Infrastructure optimization plays a vital role in effective cloud management. It involves achieving optimal performance, availability, and resource utilization while minimizing costs. Traditional infrastructure optimization techniques may not be sufficient in the context of cloud environments due to the dynamic nature of resource provisioning and fluctuating workloads. Automated resource allocation, workload balancing, and intelligent scaling mechanisms are essential components of infrastructure optimization in the cloud.

The need for multicloud deployments arises from several factors. First, organizations adopt multicloud strategies to enhance flexibility and mitigate vendor lock-in risks. By leveraging multiple cloud providers, organizations can distribute their workloads across different platforms, reducing dependence on a single vendor. Multiloud deployments offer the flexibility to choose the best-fit services from different providers, resulting in optimized performance, cost savings, and improved negotiation capabilities.

Multicloud deployments also provide advantages in terms of geographical redundancy and disaster recovery. By utilizing multiple cloud providers with data centers in different regions, organizations can ensure business continuity in the face of localized outages or natural disasters. The redundancy across multiple providers reduces the risk of service disruptions and data loss, providing robustness and reliability to critical applications.

[1] https://btech.id/news/embracing-the-sky-the-advantages-and-considerations-of-cloud-migration/
[2] https://adex.ltd/difference-between-capex-and-opex/

Furthermore, multicloud deployments enable organizations to optimize performance and workload placement. By strategically placing workloads based on factors such as latency, data sovereignty regulations, and service-level agreements (SLAs), organizations can enhance user experience and achieve optimal performance. Leveraging specialized services offered by different providers for specific use cases allows organizations to unlock the full potential of the cloud.

While existing research has explored various approaches to address these challenges, many focus on specific aspects or single-cloud environments. Some studies utilize machine learning and closed-source solutions provided by cloud vendors to forecast future billing and alert users for excessive usage. Others have used HTN (Hierarchical Task Network) planning or AI planning and combinatorial optimization techniques to create deployment plans for cloud services. However, these approaches often have limited scope, neglect the cost element, or do not consider multi-cloud scenarios.

In this thesis, we propose an approach that leverages HTN planning to optimize the cost of multi-cloud deployments post-deployment. Our approach involves modeling the infrastructure configuration and continuously monitoring infrastructure costs. We aim to develop an HTN planning model that can generate efficient reconfiguration plans across multiple clouds, considering factors such as cost and network latency between services. By utilizing AI planning techniques, we can automate the optimization process and dynamically adapt the infrastructure based on various constraints. This includes factors such as CPU idleness, high bills, and temporal shutdown of services. Additionally, we will incorporate time series statistics to forecast future resource usage and cost patterns, enabling proactive cost reduction measures.

By addressing the challenges of cloud cost reduction and infrastructure optimization in multicloud environments, this research contributes to the advancement of cloud computing. The proposed approach provides organizations with a holistic solution to enhance resource utilization, mitigate latency issues, optimize infrastructure, and manage resources effectively across multiple cloud providers. Ultimately, this enables organizations to achieve improved operational efficiency, reduced costs, and enhanced performance in their cloud-based deployments.

2 Background

This chapter revolves around the background knowledge of the areas touched on in this thesis. It is to note that due to the limited scope of this research thesis, covering in-depth knowledge is not feasible.

Computer systems commonly reside in computing clouds like Amazon Web Services, Azure, and Google Cloud Platform, offering storage, network, and computing resources through Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS) models Sowmya et al. [SDN14]. These cloud solutions reduce management effort and downtime risk while providing high scalability compared to on-premise solutions. Scalability enables the addition of new instances of services, virtual machines, or databases as needed. However, accurately predicting system load poses challenges, resulting in overprovisioning, excessive power consumption, and unnecessary expenses. To avoid emergencies, companies often provide resources with a safety margin, sometimes maintaining high levels even after resolving problems. Reducing provisioned resources is crucial for environmental protection, as data centers are projected to consume a significant portion of global electricity in the future. Cloud providers offer different components such as virtual machines (VM) and databases, each with unique properties like compute power, RAM size, disk capacity, and input/output per second (IOPS).

2.1 Cloud Computing

The advent of cloud computing has revolutionized the way businesses operate, enabling them to leverage scalable and flexible computing resources. Cloud platforms have become essential tools for organizations of all sizes, offering a wide range of services, from storage and computing power to advanced data analytics and machine learning. In the following paragraphs, we will explore the concepts of the cloud, multicloud, different types of cloud instances, regions, and the importance of adopting a multicloud strategy.

2.1.1 The Cloud

The cloud refers to a network of servers and data centers that provide on-demand access to computing resources over the internet. Cloud computing eliminates the need for local infrastructure, allowing users to access applications and store data remotely. Service providers such as Amazon Web

Services (AWS) [1], Microsoft Azure [2], and Google Cloud Platform (GCP) [3] offer various cloud services, including infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS).

2.1.2 Multicloud

Multicloud is the practice of using multiple cloud service providers to meet different business needs. Rather than relying on a single cloud provider, organizations distribute their workloads across multiple platforms. This approach provides several benefits, including mitigating vendor lock-in, enhancing resilience, optimizing costs, and leveraging the unique strengths and capabilities of each cloud provider. Multicloud also reduces the risk of service disruptions and ensures data redundancy, which is crucial for business continuity.

2.1.3 Types of Cloud Instances

Cloud instances, also known as virtual machines (VMs), are virtualized computing environments within the cloud infrastructure. They allow users to run applications and services without the need for physical servers. Cloud providers offer various instance types, each optimized for specific workloads. For example, AWS provides general-purpose instances suitable for a wide range of applications, compute-optimized instances for high-performance computing, memory-optimized instances for memory-intensive workloads, and GPU instances for accelerated computing tasks such as machine learning and graphics processing.

2.1.4 Regions

Cloud service providers have data centers located in different geographic regions worldwide. Each region consists of multiple availability zones, which are physically isolated but interconnected data centers within the same region. Customers can choose the region that best suits their needs based on factors such as latency, data sovereignty requirements, compliance regulations, and disaster recovery considerations. Deploying applications in geographically distributed regions ensures low latency, improves fault tolerance, and enables compliance with local data protection regulations.

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/en-us
[3] https://cloud.google.com/

2.1.5 Importance of Multicloud Strategy

Adopting a multicloud strategy offers several significant advantages for businesses. Firstly, it allows organizations to avoid vendor lock-in, where reliance on a single cloud provider can limit flexibility and increase costs[4]. By utilizing multiple cloud providers, companies can negotiate better pricing [5] and take advantage of unique services and features provided by each provider.

Secondly, multicloud enhances resilience and mitigates the risk of service disruptions. If one cloud provider experiences an outage [6] or service degradation, workloads can be shifted to another provider, ensuring business continuity. This redundancy reduces downtime and safeguards against potential revenue loss.

Thirdly, multicloud enables cost optimization. By distributing workloads across different providers, organizations can compare pricing, negotiate contracts, and choose the most cost-effective cloud services for each workload. Additionally, multicloud can facilitate workload placement based on factors such as data locality, regulatory compliance, and performance requirements, leading to improved resource utilization and reduced costs.

2.2 Time Series Data Analysis and Forecasting with Prophet

Time series data analysis plays a crucial role in various domains, including finance, economics, weather forecasting, and resource planning. Accurate forecasting of time series data enables organizations to make informed decisions, optimize resource allocation, and plan for the future. In recent years, the popularity of the Facebook Prophet library has soared due to its effectiveness in tackling univariate time series forecasting problems.

2.2.1 Understanding Time Series Data

Time series data refers to a collection of observations recorded at regular intervals over time. The temporal order of the data points is essential as it enables the analysis of trends, patterns, and seasonality. Time series data can exhibit various characteristics, such as trend (long-term direction), seasonality (repeating patterns), and noise (random fluctuations).

Univariate Time Series Data

Univariate time series data refers to a type of time series data that consists of a single variable observed over time at regular intervals. In other words, it involves a sequence of observations or measurements of a single phenomenon or variable. Each observation in the sequence is associated with a specific time or timestamp.

[4] https://medium.com/illuminations-mirror/pros-and-cons-of-multi-cloud-vs-single-cloud-environments-ffd0f52a10ec

[5] https://cast.ai/blog/cloud-pricing-comparison-aws-vs-azure-vs-google-cloud-platform/

[6] https://fortune.com/2023/06/07/cloud-outages-on-the-rise-tech-geopolitics-internet/

For example, consider a daily temperature dataset that records the temperature at a particular location over a certain period. In this case, the dataset contains only one variable (temperature) and its corresponding timestamps (dates). Each observation in the dataset represents a single temperature measurement at a specific time.

Univariate time series analysis focuses on understanding the patterns, trends, and seasonality within the single variable and making forecasts based on its past behavior. Techniques such as autoregressive integrated moving average (ARIMA), exponential smoothing, and Facebook Prophet can be employed for modeling and forecasting univariate time series data.

Multivariate Time Series Data

Multivariate time series data, involves multiple variables observed over time at regular intervals. It consists of a set of interrelated time series, where the values of each variable depend on its own history as well as the history of other variables in the dataset.

For instance, consider a financial dataset that includes daily stock prices, trading volumes, and interest rates for a particular set of companies over a given period. In this case, the dataset contains multiple variables (stock prices, trading volumes, interest rates) and their corresponding timestamps (dates). Each observation represents the values of these variables at a specific time.

Multivariate time series analysis involves studying the relationships and dependencies between the different variables in the dataset. It aims to capture the interactions and dynamics among the variables to gain insights and make forecasts. Methods such as Vector Autoregression (VAR), multivariate extensions of ARIMA, and machine learning algorithms like recurrent neural networks (RNNs) or long short-term memory (LSTM) networks can be utilized for modeling and forecasting multivariate time series data.

2.2.2 Introduction to Facebook Prophet

Facebook Prophet is an open-source library developed by Facebook's Core Data Science team. It aims to simplify the process of time series forecasting by providing an intuitive and flexible framework. Prophet is built on the additive model, which decomposes the time series into trend, seasonality, and holiday components. It incorporates domain knowledge and statistical methods to produce accurate and interpretable forecasts.

2.3 Automated Planning

Planning is often associated with scheduling, but it encompasses much more. In the context of intelligent agents, planning involves determining how to achieve goals by making decisions. Intelligent agents are capable of observing and interacting with the environment in a purposeful manner to achieve specific objectives. However, in order to act purposefully, the agent must have a clear objective and be able to anticipate the consequences of its actions in a given environmental state. Additionally, certain objectives are inherently complex, requiring the agent to decide which actions to take and in what sequence to execute them. This cognitive process of selecting and

organizing actions is referred to as planning [GNT04]. Automated planning, on the other hand, can be defined as the field that focuses on computational models and methods for creating, analyzing, managing, and executing plans [HLMM19].

2.3.1 Classical Planning

To provide an introductory overview, let's delve into classical planning, which represents the fundamental type of automated planning. Classical planning aims to simplify the complexities of the real world by working with an abstract representation of the environment. This approach involves making several restrictive assumptions to facilitate the planning process:

- **Finite, observable, and static:** Classical planning operates under simplified assumptions where the environment consists of a finite set of states and actions. The agent possesses complete knowledge of the current state of the environment, and no changes occur in the state unless initiated by the agent through an action.
- **Deterministic, no uncertainty:** The changes in the current state can be predicted by the agent if a given action takes place.
- **Implicit time:** There is a linear sequence of instantaneous states but no explicit model of time.

Definition 2.1.1 - Predicate
A predicate p can be used to describe the objects of the environment and their relations and is defined as p = <*symbol(p), terms(p)*> where *symbol(p)* is the predicate symbol, *terms(p)* is a set of terms, where a term can either be a constant in a finite set of constants or a variable in an infinite set of variables.

Definition 2.1.2 - Ground Predicate
Ground predicate is defined as a predicate having all the terms as constants.

Definition 2.1.3 - State
A set of ground predicates is called a state which only includes predicates that evaluate to true.

Definition 2.1.4 - Action
An action a is defined as a = <*pre(a), e(a)*> where *pre(a)* is the precondition that needs to hold to perform the action and *e(a)* is the effect of the action on the state.

Planning Domain
In order for an intelligent agent to engage in reasoning, it is essential to possess a model of the environment. This model does not necessarily need to represent every single aspect of the environment but should serve as a reliable and precise approximation.

Definition 2.1.5 - Planning Domain
A planning domain is defined as Σ = *(P, A)*, where P is a finite set of predicates and A is a finite set of actions.

In practical applications, Planning Domain Definition Languages (PDDL) are utilized to define the planning domain [AHK+98]. PDDL comes in various versions and variations, each with its own specific capabilities and limitations. In Listing 2.1, a basic example from [HLMM19] is presented that describes a simple switch and includes two actions.

Listing 2.1 PDDL domain definition example

```
(define (domain switch)
    (:requirements :strip})
    (:predicates (switch_is_on)(switch_is_off))
    (:action switch_on
        :precondition (switch_is_off)
        :effect (and (switch_is_on) (not (switch_is_off))))
    (:action switch_off
        :precondition (switch_is_on)
        :effect (and (switch_is_off) (not (switch_is_on)))))
```

Listing 2.2 PDDL problem definition example

```
(define (problem turn_it_off)
    (:domain switch)
    (:init (switch_is_on))
    (:goal (switch_is_off)))
```

Planning Problem
In order for an intelligent agent to engage in reasoning, it is necessary for it to possess an environment model. This model does not need to represent every single element of the environment, but it should provide a reliable and precise approximation.

Definition 2.1.6 - Planning Problem
A planning problem is defined as $\pi = (\Sigma, s_I, s_G)$ whhere Σ is a problem domain, s_I is the initial state and s_G is the goal state.

Like planning domains, planning problems are covered in practice using PDDL in Listing 2.2.

Definition 2.1.7 - Valid Plan
A plan is known as valid plan if the equation below holds true for planning domain Σ and a planning problem π.

$$\gamma\,(s_I,P) = s_n \in s_G$$

Planners
A Planner is responsible for taking a planning problem and devising a sequence of actions that can transform the current state of the environment into a state that fulfills the objective. Different Planners employ various problem-solving strategies. For instance, some Planners utilize theorem proving [KS+92] to generate a plan, while others rely on state-space search. Planning is a computationally challenging task [ENS95], and sometimes it is not sufficient to find any plan that accomplishes the goal. Planners may need to consider a set of metrics that determine the relative quality of different plans. This additional complexity further complicates the planning process.

2.4 Hierarchical Task Networks

Classical planning exhibits several limitations. For instance, it perceives plans as linear sequences of actions, disregarding the potential benefits of non-linear plans, as evidenced early in the research [Sac75]. By enabling a Planner to reason about non-linear plans, it becomes possible to postpone commitment to a specific action order until sufficient information is available. This approach reduces the need for exhaustive searches through all possible plan orderings. Additionally, classical Planners face significant challenges when dealing with complex domains [Wil14]. The absence of abstraction levels makes it difficult to incorporate meta-level reasoning. Although integrating expert knowledge into search algorithms is feasible, it is not a straightforward task. Similarly, steering the execution trajectory of plans is also complex.

HTN (Hierarchical Task Network) planning attempts to address these limitations by introducing additional levels of abstraction and leveraging expert knowledge to guide the planning process. HTN planning is grounded on the understanding that specifying goal states as objectives can often be unnatural, and it is more intuitive to compose abstract actions from smaller, more concrete sub-actions. Moreover, organizing domain knowledge hierarchically enables the Planner to create plans using action reduction. This approach prioritizes the consideration of the most crucial conditions initially while taking into account the finer details later in the planning process [Yan90].

HTN planning incorporates two types of constructs, namely primitive and compound tasks, to facilitate abstraction [HBBB21]. These constructs are combined to create partially ordered sets known as task networks. Primitive tasks in HTN planning are akin to actions in classical planning. They can be executed if certain conditions are met in the current environment state and have corresponding effects on the environment. On the other hand, compound tasks represent more intricate actions that cannot be executed in a single step. Instead, they need to be decomposed using specific decomposition methods. The planning domain defines these decomposition methods, each capable of breaking down a particular compound task into a task network. It is possible for multiple methods to decompose a single compound task, whereas there is a one-to-one mapping between operators and primitive tasks.

2.4.1 Formal Definitions

Over the years, there have been various efforts to establish a standardized formalism for HTN planning [BAH19]. Some of these approaches were distinctive in nature, while others shared similarities with each other. The objective of this section is not to present an exhaustive formal framework but rather to provide a formal foundation that will aid readers in understanding the underlying structure of HTN planning. The following formal structures are derived from the work of [GA14].

Definition 2.2.1 - Primitive Task
A primitive task $t_p \in T_p$ is represented as $t_p =< symbol(t_p), terms(t_p) >$, where T_p is a finite set of primitive tasks, $symbol(t_p)$ is a primitive task symbol and $terms(t_p)$ is a set of terms.

Definition 2.2.2 - Operator
An operator $o \in O$ is represented as $o =< p(o), pre(o), e(o) >$, where O is a finite set of operators, $p(o)$ is a primitive task and $pre(o)$ and $e(o)$ are precondition and effect.

Definition 2.2.3 - Compound Task
A compound task $t_c \in T_c$ is represented as $t_c =< symbol(t_c), terms(t_c) >$, where T_c is a finite set of compound tasks, $symbol(t_c)$ is a compound task symbol and $terms(t_c)$ is a set of terms.

In practical scenarios, it is often advantageous to broaden the definition of compound tasks to include a set of preconditions, denoted as $pre(t_c)$. These preconditions must be satisfied in the current world state for the Planner to consider decomposing the compound task. This extension has proven highly valuable in our implementation, particularly for compound tasks that can be decomposed by a large number of methods. While it does not offer any semantic benefits or substantial performance enhancements, it has allowed us to easily adjust the preconditions of multiple methods from a centralized location.

Definition 2.2.4 - Task Network
A task network tn is represented as $< T, <>$, where T is a finite set of tasks and $\prec$ is a partial order on T.

Definition 2.2.5 - Method
A method $m \in M$ is represented as $m =< c(m), pre(m), tn(m) >$, where M is a set of methods, $c(m)$ is a compound task and $pre(m)$ and $tn(m)$ is a precondition and task network respectively.

Definition 2.2.6 - Decomposition
For a task network $tn =< T, <>$ and a method m for a compound task $t = c(m)$, m breaks down tn into $\acute{tn}$ where $\acute{tn} = ((T/\{t\}) \cup T_m, \prec \cup \prec_m \cup \prec_D)$. The $\prec_D$ is defined as $\{(t_1, t_2) \in TxT_m | (t_1, t) \in \prec \} \cup \{(t_1, t_2) \in T_m XT | (t, t_2) \in \prec\}$.

Definition 2.2.7 - HTN Planning Domain
An HTN planning domain is represented as $\Sigma = (O, M)$, where O is a set of operators and M is a set of methods.

Definition 2.2.8 - HTN Planning Problem
An HTN planning problem network is represented as $\pi = (\Sigma, s_0, tn_0)$, where Σ is the planning domain, s_0 is the initial state and tn_0 is the initial task network.

Planners
Different HTN Planners employ various strategies to solve HTN planning problems, and they can be classified based on the search space they operate in. While this work does not aim to provide a comprehensive overview of HTN Planners as done in [BAH19] and [GA14], it offers a simplified explanation of how HTN Planners function. HTN Planners differ from classical Planners in that their objective is not to reach a specific goal state but rather to simplify a hierarchical network of tasks into a linear sequence of äctions."

Some HTN Planners utilize a decomposition-based search strategy. In this approach, applicable methods are repeatedly employed to decompose the task network until no more compound tasks remain, resulting in a task network consisting solely of primitive tasks. On the other hand, other Planners adopt a progression-based search strategy. These Planners decompose compound tasks and execute primitive tasks as applicable until the task network becomes empty. It is important to note that decomposing a compound task modifies the structure of the task network but not the state, while executing a primitive task changes the state of the environment but not the task network's structure. In a progression-based search, the Planner needs to keep track of the executed primitive tasks to avoid retracing the path from the initial task network to the empty one.

Description Languages
HTN planning domains and problems can be described using a description language, similar to classical planning. However, different Planners utilize different description languages. Nonetheless, there have been efforts to establish a standardized description language by extending PDDL to incorporate HTN constructs. A new description language called Hierarchical Domain Definition Language (HDDL), based on PDDL, has recently been adopted as the standard input language for the hierarchical planning track at the International Planning Competition. However, Planners often hesitate to adopt new formats, and currently, most Planners only support their own custom input formats.

2.5 GTPyhop Planner

GTPyhop [NBP+21] is chosen for its compatibility with the Python programming language and its ability to facilitate the development of closed-loop systems. GTPyhop is an extension of the Pyhop Planner, which is a straightforward SHOP-style Planner implemented in Python. Pyhop was not heavily promoted, but its user-friendly nature and simplicity made it a popular choice in various projects and research publications.

GTPyhop, an extension of Pyhop, merges the task decomposition approach of SHOP-style Planners with the goal decomposition approach of GDP-style Planners. This combination enables planning for both goals and tasks, offering flexibility in representing objectives in various forms. Compared to Pyhop, GTPyhop has a larger source code base and incorporates additional functionalities. These include the ability to load multiple planning domains, and switch between them without restarting Python, as well as improved documentation and debugging features.

The planning algorithm of GTPyhop builds upon a depth-first search strategy and expands upon Pyhop by incorporating refinements for goal decomposition as well as a combination of task and goal decomposition. The algorithm guarantees the soundness of the generated plan by verifying whether it accomplishes the decomposed goal and backtracking if required. The GTPyhop provides pseudocode for the planning algorithm of GTPyhop.

The GTPyhop provides a detailed explanation of the representations and examples employed in GTPyhop planning domains. In GTPyhop, domains are represented as Python objects, encompassing the various elements of the planning domain. States are also represented as Python objects, serving as collections of state-variable bindings. Actions and methods are implemented as Python functions, utilizing Python if tests and computations to define preconditions and effects. Tasks and task methods are represented as tuples, while goals can be represented as either unigoals or multigoals.

We take the blocks-world usecase as an example from the GTPyhop paper [NBP+21]. Blocks-world involves manipulating blocks on a table. We'll use this usecase to explain the concepts mentioned in the paper.

2.5.1 Representation of States

In GTPyhop, states are represented using Python objects. For example, consider the initial state sus_s0 in the blocks-world domain.

```
sus_s0.pos = {'a': 'table', 'b': 'table', 'c': 'a'}
sus_s0.clear = {'a': False, 'b': True, 'c': True}
sus_s0.holding = {'hand': False}
```

This state represents the initial configuration where blocks *a* and *b* are on the table, and block *c* is on top of block *a*. The *clear* attribute indicates whether a block is clear or not, and the *holding* attribute indicates if the robot hand is holding any block.

2.5.2 Action and Methods

Actions and methods are defined as Python functions. Let's consider the action *pickup* that picks up a block. Its definition would look like the following.

```
def pickup(s, x):
    if s.pos[x] == 'table' and s.clear[x] == True and s.holding['hand'] == False:
        s.pos[x] = 'hand'
        s.clear[x] = False
        s.holding['hand'] = x
        return s
```

This action checks the preconditions (e.g., the block is on the table, clear, and the hand is empty) and modifies the state accordingly.

2.5.3 Tasks and Task Methods

Tasks represent goals or sub-goals in the planning process. Let's consider the task *take*, which involves picking up a block, which is covered in the code below.

```
def take(s, x):
    if s.clear[x] == True:  # precondition
        if s.pos[x] == 'table':  # decide what to do
            return [('pickup', x)]
        else:
            return [('unstack', x, s.pos[x])]
```

This method decomposes the *take* task into sub-tasks based on the state. If the block is clear, it either returns a subtask to *pickup* the block from the table or *unstack* it from another block.

2.5.4 Goals and Goal Methods

Goals represent the desired state or conditions to be achieved. In GTPyhop, goals can be represented as unigoals (a desired value for a state variable) or multigoal (a conjunction of unigoals). Here is a multigoal example.

```
sus_sg.pos = {'a': 'b', 'b': 'c'}
```

This multigoal specifies that block a should be on top of block b, and block b should be on top of block c.

To achieve this goal, we can define a goal method that decomposes the multigoal into subgoals and corresponding actions or tasks. The paper [NBP+21] provides an example of a goal method for the blocks-world domain, which is omitted here for brevity.

2.5.5 Planning Algorithm

The GTPyhop planning algorithm follows a depth-first search (DFS) approach. It recursively explores the planning space by decomposing tasks and goals using methods until a solution plan is found or deemed impossible.

3 Related Work

In the domain of multi-cloud deployments, several studies have focused on optimizing deployment plans using different techniques and perspectives. At the component level composition of the web service to deploy in a single cloud is done in [GNLA17]. They use HTN planning to make deployment plans for particular web services and use. Their modeling is only limited to the deployment of multiple components of an application in a single cloud. They didn't take into account the cost element and multi-cloud scenario.

A group of researchers in [ZCY+10] use AI planning and combinatorial optimization to create a deployment plan of dependent cloud service in multi-cloud by optimizing cost. They used combinatorial optimization to find low-cost cloud providers from a set of cloud providers and create a deployment plan afterward. However, their work is only limited to one-time deployment. They are not considering post-deployment cost optimization where in our solution we consider post deployment optimization of service in multicloud scenarios.

Another study [SAGS14] addresses automatic scaling of cloud resources to reduce costs for clients. The authors propose a model that organizes resources in a cloud environment based on service requests. They employ clustering and analysis of request patterns to estimate the number of virtual machines needed and configure services accordingly. However, their load metric is limited and does not account for multicloud scenarios and doesn't have modern cloud dynamics and complexity.

In [ON20], a novel approach for optimizing cloud resource usage costs using anomaly detection, machine learning, and particle swarm optimization is presented. Although it offers a closed-loop solution that adapts to changing loads and pricing plans, it lacks support for reconfiguration of existing services, moving across regions, and considering different domain knowledge. Multicloud scenarios are also not considered, making the configuration process more complex.

The approach presented in [MH11] focuses on auto-scaling cloud workflows to minimize cost and meet application deadlines. The authors propose a dynamic algorithm that allocates and deallocates virtual machines and schedules tasks on the most cost-efficient instances based on workload and resource availability. However, the approach has limitations, such as relying on users to provide job deadlines, not considering long-running instances, and lacking consideration of other factors during the optimization and rescaling of virtual machines.

In [YXJ+18] and [ANE13], a solution is presented that analyzes incoming tasks and allocates virtual machine instances in a cost-effective manner while meeting deadlines. However, this solution requires knowledge of the CPU and memory requirements of the tasks. In contrast, our proposed solution focuses on infrastructure optimization rather than task-based optimization, considering various factors to ensure sufficient resources are available to meet system requirements.

The significance of pricing models in cloud computing services has been addressed in previous research, such as the work presented in [WC15], which focuses on the issue of service bundling. The paper introduces a model aimed at assisting customers in identifying the most cost-effective

combination of service providers. However, our approach goes beyond relying solely on pricing models by taking into account multiple factors, including pricing, to develop a comprehensive solution.

Our study proposes an approach that leverages HTN planning to optimize the cost of multi-cloud deployments post-deployment. The approach considers both cost and network latency as optimization objectives, utilizes HTN planning for expressive and scalable reconfiguration plans, employs time series statistical models to predict future virtual machine status, and supports multi-cloud scenarios with heterogeneous cloud providers.

4 System Design

This chapter covers the architecture of our proposed solution and the elaboration of its components. Moreover, we have covered the functional and non-functional requirements of the system.

4.1 Functional Requirements

1. **Data Capture:** The system should be able to capture real-time cloud data from various sources, including resource usage, cost, and infrastructure changes.
2. **Problem File Generation:** The system should generate problem files that can be used by the Planner to generate optimized plans.
3. **Planner Functionality:** The Planner should utilize the GTPyhop HTN Planner to generate optimized plans based on the problem files generated from the Cloud Simulator and Learner module.
4. **Execution of Plans:** The executor should be able to execute the plans generated by the Planner by communicating with the Cloud Simulator through API calls.
5. **Load Prediction:** The Learner module should use the FB Prophet time series library to predict future system load as a heuristic for identifying virtual machines that can be turned off to optimize cost.
6. **Visualization:** The system should employ Kibana[1] with Elasticsearch[2] to provide a user-friendly interface for visualizing metrics, system, and output.

4.2 Non-Functional Requirements

1. **Performance:** The system should be able to handle large volumes of data efficiently and generate optimized plans in a timely manner.
2. **Scalability:** The system should be designed to scale and accommodate increasing data and infrastructure complexity.
3. **Reliability:** The system should be reliable, with minimal downtime and the ability to recover from failures.

[1] https://www.elastic.co/kibana/
[2] https://www.elastic.co/elastic-stack/

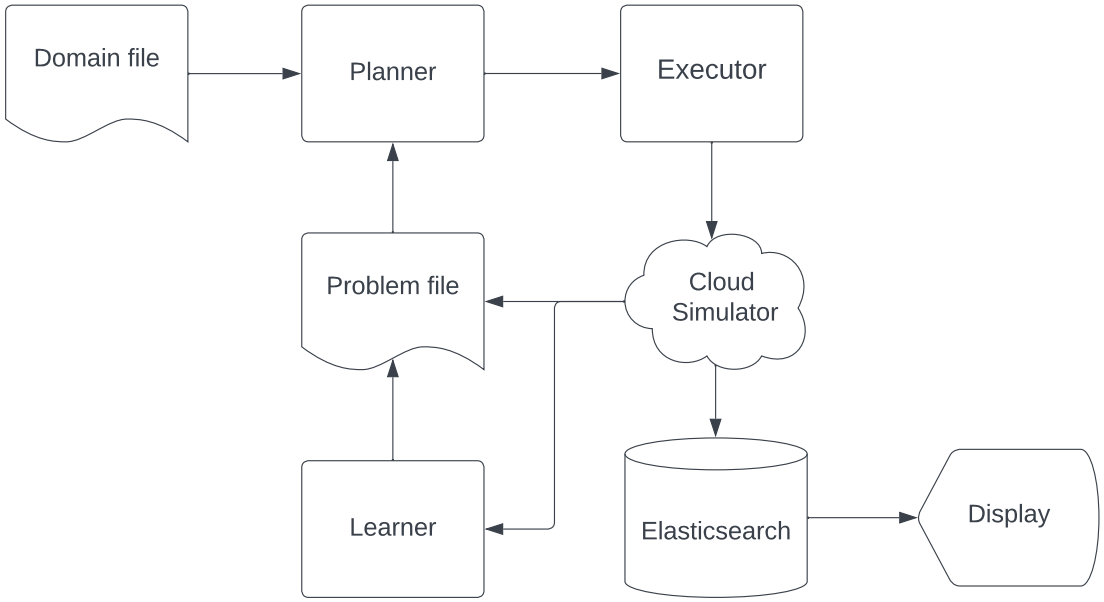

Figure 4.1: Architecture of our proposed solution

4. **Usability:** The user interface should be intuitive and easy to navigate, allowing users to interact with the system's output.

4.3 Architecture Diagram

In Figure 4.1, we have constructed an overview of our proposed solution consisting of multiple components which are explained below.

Planner: The Planner plays a crucial role in our system as it utilizes the GTPyhop HTN Planner. By using the Python version of the simple shop Planner, we eliminate the need to call an external Planner via a shell. This approach provides us with more flexibility in defining our domain and problem files. The Planner is responsible for generating optimized plans based on the given constraints and goals, enabling efficient resource allocation and system optimization.

Executor: The executor module is of great importance as it is responsible for executing the plans generated by the Planner. It acts as the bridge between the Planner, the Cloud Simulator Microservice, and Elasticsearch. The executor communicates with the Cloud Simulator and Elasticsearch to perform the necessary actions defined in the plan. This includes calling the appropriate APIs and executing update queries to modify the state of the cloud infrastructure. The executor ensures the seamless execution of each action in the plan, contributing to the effective implementation of the optimization strategies.

Learner: The Learner module, which incorporates machine learning techniques, holds significance in predicting the future system load for different virtual machines. By accurately forecasting the system load, we gain valuable insights into resource utilization and performance trends. Although currently focused on load prediction, the Learner module can be extended to encompass other

metrics such as hard disk or RAM, providing a comprehensive understanding of the system's behavior. The output of the Learner module serves as a heuristic for identifying virtual machines that can be turned off, leading to efficient resource management and cost optimization.

Cloud Simulator: The Cloud Simulator is a fundamental component in our system, simulating the behavior of a cloud environment. It creates a realistic and controlled environment for testing and executing the plans generated by the Planner. By mimicking the characteristics and functionalities of a real cloud infrastructure, the simulator enables accurate assessment and validation of the optimization strategies. It plays a pivotal role in evaluating the feasibility and effectiveness of the generated plans before moving implementation to a production environment.

Elasticsearch: Elasticsearch serves as a vital component in our system, acting as a powerful search and analytics engine. Its capabilities in storing, retrieving, and analyzing large volumes of data make it invaluable for managing the data generated by our system. By indexing and searching various data sources, Elasticsearch enables efficient data management and facilitates quick retrieval of information. It empowers us to effectively store and query the state of the cloud infrastructure, the executed plans, and other relevant data and enables comprehensive analysis.

Display: The display component enhances the usability of our system by presenting information and visualizations to users. It provides a user-friendly interface that allows users to view and interact with the results, forecasts, and other pertinent data generated by our system. The display component plays a crucial role in facilitating effective communication of the system's findings and recommendations, enabling users to make informed decisions based on the presented insights.

5 Implementation

It is observed that building a system that optimizes cloud cost and infrastructure via automated planning is a daunting task. Building such a solution requires real-time cloud data in a stream to simulate the effectiveness of a Planner. Moreover, capturing cloud infrastructure context and mimicking the future workload to generate problem files for Planners is a task in itself. Another problem is observing all these changes and cloud infrastructure monitoring via dashboards like those provided by AWS, Azure, and GCP. For this purpose, we have introduced a robust solution that takes into consideration these problems to overcome them.

In the beginning, there is a Cloud Simulator component that generates synthetic data and simulates cloud behavior for different actions. This data serves as input to the Learner component, which predicts future virtual machine CPU load using the FB Prophet time series library for univariate forecasting.

The domain is modeled using GTPyhop in Python, and a problem file is generated based on the context provided by the Cloud Simulator and Learner module. The output from the Learner module is then fed into the executor, which executes actions using API calls provided by the simulator.

To visualize metrics and results, Kibana with Elasticsearch is employed. Different graphs are created to display the Cloud Simulator's state and cost dashboard. Elasticsearch is used to store synthetic data and metadata, including information such as the last plan executed and other important details. It helps us in evaluation of results as we can simulate load in realtime and the whole system in a closed loop.

Furthermore in this chapter, we will discuss the infrastructure modeling followed by how the Cloud Simulator utilizes that infrastructure to generate data and other key features of the simulator. We will then delve into the Learner module and its utilization of the Cloud Simulator data. Subsequently, we will explain the domain modeling in GTPyhop and how the problem file is generated. After covering the Planner module, we will briefly touch the executioner module, demonstrating how actions are executed on the Cloud Simulator. Finally, we will explore the Kibana and Elasticsearch modules, highlighting their crucial role in linking all the components together.

5.1 Cloud Infrastructure Modeling

To model our cloud infrastructure allocation formally, we can represent each Virtual Machine (VM) as a node in a graph, with the dependencies between VMs represented as edges connecting the nodes. The following mathematical representation describes the variables and their meanings.

V: A set of nodes representing VMs
E: A set of edges representing dependencies between VMs
c(i): The cloud provider of VM_i

r(i): The region of VM_i
A(i): The set of applications running on VM_i
L(i, j): The latency between VM_i and VM_j in milliseconds
S(i): The instance size of VM_i
C(i): The cost per hour for running VM_i
I(i): The inbound traffic of VM_i
O(i): The outbound traffic of VM_i
D(i): The hard disk consumption of VM_i
T(i): The caching enabled status of VM_i
ST(i): The storage capacity of VM_i
RF(i): The replication factor for VM_i

We have this data which we will represent as the variables mentioned above.

V = {vm_aws_1, vm_aws_2, vm_aws_3, vm_aws_4, vm_aws_5, vm_azure_1, vm_azure_2, vm_azure_3, vm_azure_4, vm_gcp_1, vm_gcp_2, vm_gcp_3, vm_gcp_4, db_aws_1, db_azure_1, db_gcp_1}

E = {(vm_aws_1, vm_aws_2), (vm_aws_1, db_aws_1), (vm_aws_3, vm_aws_2), (vm_aws_5, vm_aws_2), (vm_azure_1, vm_azure_2), (vm_azure_1, db_azure_1), (vm_azure_3, vm_azure_2), (vm_gcp_1, vm_gcp_2), (vm_gcp_1, db_gcp_1), (vm_gcp_3, vm_gcp_2)}

$c(\text{vm\_aws}_1)$[1] = "aws"
$c(\text{vm\_aws}_2)$ = "aws"
$c(\text{vm\_aws}_3)$ = "aws"
$c(\text{vm\_aws}_4)$ = "aws"
$c(\text{vm\_aws}_5)$ = "aws"
$c(\text{vm\_azure}_1)$ = "azure"
$c(\text{vm\_azure}_2)$ = "azure"
$c(\text{vm\_azure}_3)$ = "azure"
$c(\text{vm\_azure}_4)$ = "azure"
$c(\text{vm\_gcp}_1)$ = "gcp"
$c(\text{vm\_gcp}_2)$ = "gcp"
$c(\text{vm\_gcp}_3)$ = "gcp"
$c(\text{vm\_gcp}_4)$ = "gcp"
$c(\text{db\_aws}_1)$ = "aws"
$c(\text{db\_azure}_1)$ = "azure"
$c(\text{db\_gcp}_1)$ = "gcp"

$r(\text{vm\_aws}_1)$[2] = "us-west-1"
$r(\text{vm\_aws}_2)$ = "us-west-1"
$r(\text{vm\_aws}_3)$ = "eu-west-1"
$r(\text{vm\_aws}_4)$ = "ap-southeast-1"
$r(\text{vm\_aws}_5)$ = "ap-southeast-1"
$r(\text{vm\_azure}_1)$ = "westeurope"
$r(\text{vm\_azure}_2)$ = "westeurope"
$r(\text{vm\_azure}_3)$ = "southeastasia"

[1] *c(vm)* represents the cloud provider of the VM
[2] *r(vm)* represents the region of the VM

$r(vm\_azure_4) = "southeastasia"$
$r(vm\_gcp_1) = "us\text{-}central\text{-}2"$
$r(vm\_gcp_2) = "us\text{-}central\text{-}1"$
$r(vm\_gcp_3) = "asia\text{-}east\text{-}1"$
$r(vm\_gcp_4) = "asia\text{-}east\text{-}1"$
$r(db\_aws_1) = "us\text{-}west\text{-}1"$
$r(db\_azure_1) = "westeurope"$
$r(db\_gcp_1) = "us\text{-}central\text{-}1"$

$a(vm\_aws_1)$[3] $= ["nginx"]$
$a(vm\_aws_2) = ["app"]$
$a(vm\_aws_3) = ["app", "python\ microservice", "java\ microservice"]$
$a(vm\_aws_4) = ["web"]$
$a(vm\_aws_5) = ["nginx"]$
$a(vm\_azure_1) = ["web", "python\text{-}microservice", "java\text{-}microservice"]$
$a(vm\_azure_2) = ["app"]$
$a(vm\_azure_3) = ["web"]$
$a(vm\_azure_4) = ["app"]$
$a(vm\_gcp_1) = ["app"]$
$a(vm\_gcp_2) = ["web"]$
$a(vm\_gcp_3) = ["nginx"]$
$a(vm\_gcp_4) = ["app", "sql"]$
$a(db\_aws_1) = ["sql"]$
$a(db\_azure_1) = ["sql"]$
$a(db\_gcp_1) = ["sql"]$

$l(vm\_aws_1, vm\_aws_2)$[4] $= 5$
$l(vm\_aws_1, db\_aws_1) = 15$
$l(vm\_aws_3, vm\_aws_2) = 8$
$l(vm\_aws_5, vm\_aws_2) = 10$
$l(vm\_azure_1, vm\_azure_2) = 6$
$l(vm\_azure_1, db\_azure_1) = 12$
$l(vm\_azure_3, vm\_azure_2) = 12$
$l(vm\_gcp_1, vm\_gcp_2) = 8$
$l(vm\_gcp_1, db\_gcp_1) = 18$
$l(vm\_gcp_3, vm\_gcp_2) = 10$

$s(vm\_aws_1)$[5] $= "t2.micro"$
$s(vm\_aws_2) = "t3.small"$
$s(vm\_aws_3) = "m5.large"$
$s(vm\_aws_4) = "t3.nano"$
$s(vm\_aws_5) = "m5.xlarge"$
$s(vm\_azure_1) = "Standard\_B1s"$
$s(vm\_azure_2) = "Standard\_D2s\_v3"$

[3] *a(vm)* represents the set of applications running on the VM
[4] *l(vm₁, vm₂)* represents the latency between VM_1 and VM_2 in milliseconds
[5] *s(vm)* represents the instance size of the VM

$s(vm\_azure_3) = \text{"Standard\_F2s\_v2"}$
$s(vm\_azure_4) = \text{"Standard\_DS1\_v2"}$
$s(vm\_gcp_1) = \text{"f1-micro"}$
$s(vm\_gcp_2) = \text{"n1-standard-1"}$
$s(vm\_gcp_3) = \text{"g1-small"}$
$s(vm\_gcp_4) = \text{"n1-highcpu-2"}$
$s(db\_aws_1) = \text{"db.t2.micro"}$
$s(db\_azure_1) = \text{"Standard\_D2s\_v3"}$
$s(db\_gcp_1) = \text{"db-f1-micro"}$

$c(vm\_aws_1)$[6] $= 0.02$
$c(vm\_aws_2) = 0.05$
$c(vm\_aws_3) = 0.15$
$c(vm\_aws_4) = 0.01$
$c(vm\_aws_5) = 0.20$
$c(vm\_azure_1) = 0.03$
$c(vm\_azure_2) = 0.08$
$c(vm\_azure_3) = 0.12$
$c(vm\_azure_4) = 0.05$
$c(vm\_gcp_1) = 0.01$
$c(vm\_gcp_2) = 0.04$
$c(vm\_gcp_3) = 0.02$
$c(vm\_gcp_4) = 0.07$
$c(db\_aws_1) = 0.03$
$c(db\_azure_1) = 0.05$
$c(db\_gcp_1) = 0.02$

$i(vm\_aws_1)$[7] $= 0$
$i(vm\_aws_2) = 0$
$i(vm\_aws_3) = 0$
$i(vm\_aws_4) = 0$
$i(vm\_aws_5) = 0$
$i(vm\_azure_1) = 0$
$i(vm\_azure_2) = 0$
$i(vm\_azure_3) = 0$
$i(vm\_azure_4) = 0$
$i(vm\_gcp_1) = 0$
$i(vm\_gcp_2) = 0$
$i(vm\_gcp_3) = 0$
$i(vm\_gcp_4) = 0$
$i(db\_aws_1) = 0$
$i(db\_azure_1) = 0$
$i(db\_gcp_1) = 0$

[6] *c(vm)* represents the cost per hour for running the VM
[7] *i(vm)* represents the inbound traffic of the VM

$o(vm\_aws_1)^{8} = 0$
$o(vm\_aws_2) = 0$
$o(vm\_aws_3) = 0$
$o(vm\_aws_4) = 0$
$o(vm\_aws_5) = 0$
$o(vm\_azure_1) = 0$
$o(vm\_azure_2) = 0$
$o(vm\_azure_3) = 0$
$o(vm\_azure_4) = 0$
$o(vm\_gcp_1) = 0$
$o(vm\_gcp_2) = 0$
$o(vm\_gcp_3) = 0$
$o(vm\_gcp_4) = 0$
$o(db\_aws_1) = 0$
$o(db\_azure_1) = 0$
$o(db\_gcp_1) = 0$

$d(vm\_aws_1)^{9} = 0$
$d(vm\_aws_2) = 0$
$d(vm\_aws_3) = 0$
$d(vm\_aws_4) = 0$
$d(vm\_aws_5) = 0$
$d(vm\_azure_1) = 0$
$d(vm\_azure_2) = 0$
$d(vm\_azure_3) = 0$
$d(vm\_azure_4) = 0$
$d(vm\_gcp_1) = 0$
$d(vm\_gcp_2) = 0$
$d(vm\_gcp_3) = 0$
$d(vm\_gcp_4) = 0$
$d(db\_aws_1) = 50$
$d(db\_azure_1) = 50$
$d(db\_gcp_1) = 50$

$t(vm\_aws_1)^{10} = 0$
$t(vm\_aws_2) = 0$
$t(vm\_aws_3) = 0$
$t(vm\_aws_4) = 0$
$t(vm\_aws_5) = 0$
$t(vm\_azure_1) = 0$
$t(vm\_azure_2) = 0$
$t(vm\_azure_3) = 0$
$t(vm\_azure_4) = 0$

[8] *o(vm)* represents the outbound traffic of the VM
[9] *d(vm)* represents the hard disk consumption of the VM in gigabytes
[10] *t(vm)* represents the caching enabled status of the VM

$t(vm\_gcp_1) = 0$
$t(vm\_gcp_2) = 0$
$t(vm\_gcp_3) = 0$
$t(vm\_gcp_4) = 0$
$t(db\_aws_1) = 0$
$t(db\_azure_1) = 0$
$t(db\_gcp_1) = 0$

$st(vm\_aws_1)$[11] $= 10$
$st(vm\_aws_2) = 20$
$st(vm\_aws_3) = 30$
$st(vm\_aws_4) = 10$
$st(vm\_aws_5) = 50$
$st(vm\_azure_1) = 10$
$st(vm\_azure_2) = 20$
$st(vm\_azure_3) = 30$
$st(vm\_azure_4) = 10$
$st(vm\_gcp_1) = 10$
$st(vm\_gcp_2) = 20$
$st(vm\_gcp_3) = 30$
$st(vm\_gcp_4) = 10$
$st(db\_aws_1) = 100$
$st(db\_azure_1) = 100$
$st(db\_gcp_1) = 100$

$rf(vm\_aws_1)$[12] $= 1$
$rf(vm\_aws_2) = 1$
$rf(vm\_aws_3) = 1$
$rf(vm\_aws_4) = 1$
$rf(vm\_aws_5) = 1$
$rf(vm\_azure_1) = 1$
$rf(vm\_azure_2) = 1$
$rf(vm\_azure_3) = 1$
$rf(vm\_azure_4) = 1$
$rf(vm\_gcp_1) = 1$
$rf(vm\_gcp_2) = 1$
$rf(vm\_gcp_3) = 1$
$rf(vm\_gcp_4) = 1$
$rf(db\_aws_1) = 3$
$rf(db\_azure_1) = 3$
$rf(db\_gcp_1) = 3$

Above shows how we model our infrastructure via graph model. The modeling is done for a sample of data we are using. This is extendable and can include 1000s of virtual machines and databases. This data is stored in elasticsearch with its mapping to resources and region.

[11] *st(vm)* represents the storage capacity of the VM in gigabytes
[12] *rf(vm)* represents the replication factor for the VM

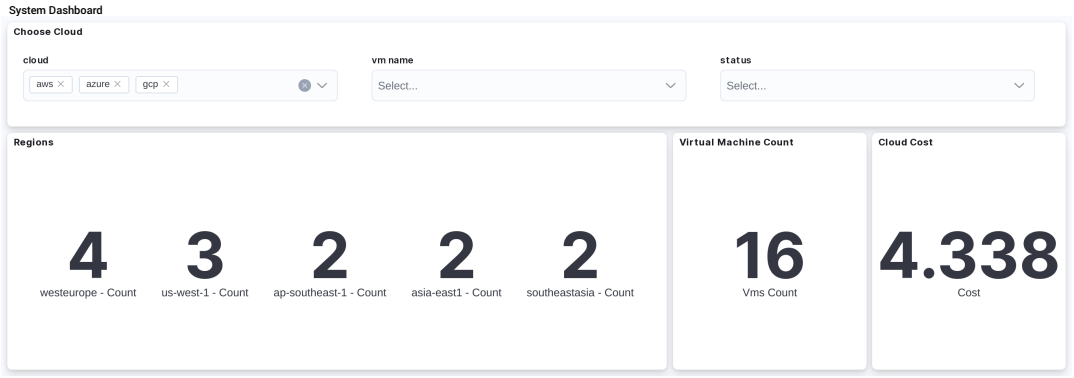

Figure 5.1: Dashboard of proposed infrastructure

In Figure 5.1, it can be seen that user can select Clouds, VMs present in those clouds and their status. After selection user can see the regions of the VMs, their count and the total cloud cost.

5.2 Cloud Simulator

The Cloud Simulator engine handles various operations required for successfully simulating real cloud scenarios. It allows users to perform actions such as adding or deleting virtual machines, and querying the state and data of virtual machines.

Data generation occurs continuously in the background task. The infrastructure schema is read from Elasticsearch, and random data is generated following a uniform random distribution. Every 60 seconds, a new record is added to indicate the VM's activity during the past minute. Each record contains information such as CPU usage, RAM usage, latency, hard disk usage, inbound and outbound traffic, and the load of each service running within the virtual machine.

Additionally, users can create, delete, and update specific virtual machines through the simulator's VM interface.

Cloud simulator engine provides multiple key features which are mentioned below.

1. **VM Generation and Management:** Creation and deletion of virtual machines within the simulated cloud infrastructure, with various attributes and properties.
2. **API Interface:** Exposure of API endpoints using the FastAPI framework, enabling users to interact with the simulator for actions like creating, deleting, and modifying virtual machines, retrieving VM information and cloud load data, and managing rules for VM behavior.
3. **Background Tasks and Simulation:** Utilization of background tasks and asynchronous programming to simulate VM behavior and generate random load values for different attributes, considering special rules defined for specific VMs.
4. **Elasticsearch Integration:** Leverages Elasticsearch to store and retrieve VM-related data, allowing easy retrieval and analysis of VM information and load statistics.

Figure 5.2: Dashboard of proposed infrastructure

5. **Rule-based VM Behavior:** Provision of support to definition and application of rules that control VM behavior, stored in Elasticsearch and accessible through the API endpoints. These rules influence VM load generation based on specific conditions.

By combining these features, the Cloud Simulator engine enables users to effectively simulate and manage virtual machines within a cloud infrastructure. It provides an API-driven interface for performing VM operations and offers simulated VM load data for testing and analysis purposes.

In Figure 5.2, we have our dashboard of Virtual Machines where it can be seen that data is being simulated.

A sample code of the domain and problem file (initial state) is given in A.1 and A.2

5.3 Learner

To implement our Learner module, we utilized a Cloud Simulator that provided synthetic per-minute data reflecting the resource usage of a cloud environment. This allowed us to extract detailed information about CPU load, which played a crucial role in predicting future resource requirements and optimizing costs.

The first step in our implementation was to query the data for a particular virtual machine (VM) using an Elasticsearch query. This allowed us to retrieve the historical CPU load data specific to that VM.

Next, we trained the FB Prophet forecasting model using the extracted data. FB Prophet is a robust time series forecasting algorithm known for capturing underlying patterns and trends in the data. By analyzing the historical CPU load data, FB Prophet generated accurate predictions of future CPU load values for the VM.

The predicted CPU load values were instrumental in our cost optimization efforts. By utilizing these future values, we could determine which virtual machines needed to be turned off to avoid unnecessary costs. It's important to note that the forecasting algorithm had to be run separately for each virtual machine, as each VM has its own workload history. Mixing data from different VMs would have compromised the accuracy of the predictions.

The forecasting process involved querying the data, training the model, and transforming the data for future prediction. We configured the prediction settings to determine the time range for forecasting, whether it was for the next four hours or four weeks.

By incorporating accurate predictions of CPU load, our implementation allowed us to make informed decisions about virtual machine management. This approach enabled us to optimize costs by identifying opportunities to power down or scale virtual machines based on their forecasted resource requirements.

5.4 Planner

The Planner module takes a problem file as input and generates a plan that will be passed to the executor module. We use GTPyhop to model the domain and facilitate the planning process. In our optimization approach, we focus on key cloud services, specifically virtual machines, virtual machine costs, incoming traffic via load balancers (such as Nginx), latency between dependent services, databases, and archival of databases.

5.4.1 Optimizing Cloud Infrastructure

One aspect of optimization involves planning how to efficiently move virtual machines and databases between regions to minimize latency. It is crucial to ensure that dependent services and databases are located close to each other to reduce round trip time, especially in microservice architectures where there can be dependencies between applications.

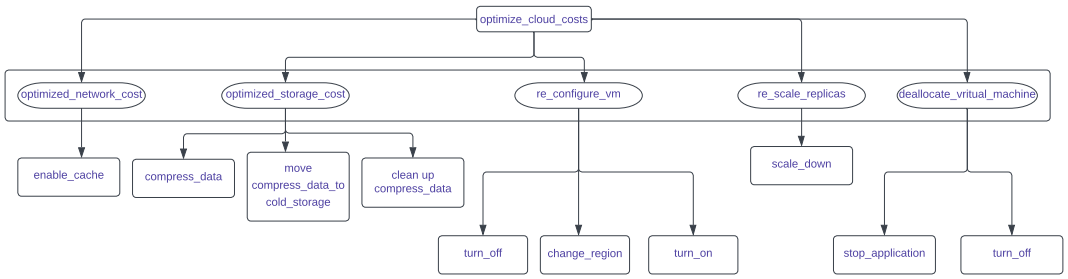

Figure 5.3: Domain Planning

By considering the network I/O or outbound traffic of virtual machines, we can enable caching to avoid redundant downloads and improve response time. For example, GitLab and Bitbucket utilize this caching mechanism in their CI/CD pipelines to optimize build times. Storage optimization is another crucial factor. By compressing data and storing it in remote locations like S3 (cold storage), we can avoid attaching additional hard disks. This usecase is particularly relevant for streaming applications, such as monitoring applications that receive terabytes of data daily. Since this data often becomes irrelevant after a day, moving it to cold storage frees up disk space for new data.

5.4.2 Reducing Cloud Bills

Since we follow a pay-as-you-go model, every second counts when it comes to cost optimization. While numerous services can contribute to reducing infrastructure costs, we focus on the virtual machine service, given its widespread usage. To decide whether to turn off a virtual machine, we employ time-series forecasting techniques that leverage historical data to predict future loads. This enables us to determine the impact of turning off a virtual machine on other services. Additionally, enabling cache helps reduce outbound requests and is particularly relevant for cloud providers like Hetzner, which have limited outbound traffic quotas. By leveraging caching, we can optimize costs by staying within these limits.

Domain planning in sudo format is stated below while basic decomposition of methods into primitive tasks can be seen in Figure 5.3.

Methods:

1. optimize_cloud_costs(vm) :
    - Preconditions: optimize_cloud_costs(vm) is applicable if the VM's status is "on" and its future status is also "on".
    - Subtasks: [("optimized_network_cost", vm), ("optimized_storage_cost", vm), ("re_configure_vm", vm), ("re_scale_replicas", vm)].
2. optimize_cloud_costs(vm) :

- Preconditions: optimized_network_cost(vm) is applicable if the average outbound network traffic of the VM is greater than 1100 requests per second and caching is not enabled.
- Subtasks: [("enable_cache", vm)].

3. optimized_storage_cost(vm) :
   - Preconditions: optimized_storage_cost(vm) is applicable if the average hard disk usage of the VM is greater than or equal to 80.
   - Subtasks: [("compress_data", vm), ("move_compress_data_to_cold_storage", vm), ("clean_up_compress_data", vm)].
4. deallocate_virtual_machine(vm) :
   - Preconditions: None.
   - Subtasks: [("stop_applications", vm), ("turn_off", vm)].
5. re_configure_vm(vm) :
   - Preconditions: re_configure_vm(vm) is applicable if the VM is not turned off and it has a dependency on a database.
   - Subtasks: [("turn_off", vm), ("change_region", vm, db_region), ("turn_on", vm)].
6. re_scale_replicas(vm) :
   - Preconditions: re_scale_replicas(vm) is applicable if the VM is of type "db", its average CPU usage is less than 40, its future status is "on", and the replication factor is greater than 1.
   - Subtasks: [("scale_down", vm)].
7. stop_applications(vm) :
   - Preconditions: stop_applications(vm) is applicable if there are applications installed on the VM.
   - Subtasks: [("stop_application", vm)].

Operations (Actions):

1. turn_off(state, vm):
   - Preconditions: The VM's current status is "on".
   - Effects: The VM's status is changed to "off".
2. turn_on(state, vm):
   - Preconditions: The VM's current status is "off".
   - Effects: The VM's status is changed to "on".
3. scale_down(state, vm):
   - Preconditions: The replication factor of the VM is greater than 1.

- Effects: The replication factor of the VM is reduced to 1.

4. change_region(state, vm, new_region):
    - Preconditions: The VM's current region is different from the new region, and the VM is turned off.
    - Effects: The VM's region is changed to the new region.
5. enable_cache(state, vm):
    - Preconditions: Caching is currently disabled for the VM.
    - Effects: Caching is enabled for the VM.
6. compress_data(state, vm):
    - Preconditions: The flag for compressing data is not set for the VM.
    - Effects: The flag for compressing data is set for the VM.
7. move_compress_data_to_cold_storage(state, vm):
    - Preconditions: The flag for moving compressed data to cold storage is not set for the VM.
    - Effects: The flag for moving compressed data to cold storage is set for the VM.
8. clean_up_compress_data(state, vm):
    - Preconditions: The flag for cleaning up compressed data is not set for the VM.
    - Effects: The flag for cleaning up compressed data is set for the VM.
9. create_snapshot(state, vm):
    - Preconditions: None.
    - Effects: A snapshot of the VM is created.
10. stop_application(state, vm):
    - Preconditions: There is an application running on the VM.
    - Effects: The application running on the VM is stopped.

5.5 Executioner

The executor module plays a crucial role in implementing the actions defined in the plan generated by the Planner module. It receives the plan as input and applies the necessary actions to modify the state of the cloud infrastructure.

The plan is provided to the executor module as a list of tuples. Each tuple represents an action to be executed and consists of an action name followed by the cloud and virtual machine names involved in the action. In some cases, additional parameters may be included. For example, consider the tuple '(change_region, 'vm_aws5', westeurope)'. In this case, the third argument 'westeurope' represents the new region to which the 'vm_aws5' virtual machine needs to be migrated.

The executor module processes each action in the plan and applies the necessary changes to the cloud infrastructure. It utilizes the same set of APIs discussed earlier to interact with the simulator and modify the state of the virtual machines and other components.

The executor module has the following steps in order to apply action successfully

1. Collect necessary data for VMs
    - Iterate through the actions and collect unique VM names.
    - Retrieve the VM ID and record from a data source
    - Store the VM data in the 'vm_data' dictionary, where the key is the VM name and the value is a tuple containing the VM ID and record
2. Apply actions to VM data
    - Iterate over the actions and apply the corresponding changes to the VM data in the 'vm_data dictionary'.
    - For example, if the action is "turn_off", set the VM's status to "off". If the action is "change_region", update the VM's region, and so on.
3. Update VM data in Elasticsearch or relevant data store
    - Iterate through the 'vm_data' dictionary and update the VM records in Elasticsearch or the relevant data store.
    - Formulate an update query using the VM record and update the corresponding document in the index.

5.6 Monitoring

After applying the plan to the cloud infrastructure, it becomes crucial to monitor the system's performance and track any changes that occur over time. This monitoring process is facilitated through the use of Elasticsearch and Kibana, which provide effective visualization and monitoring capabilities.

As the plan's actions are executed, relevant data about the infrastructure, including the VMs and their attributes, is stored in Elasticsearch. This continuous updating of the data store ensures that the current state of the cloud infrastructure is accurately reflected. Information such as VM statuses (on/off), regions, caching settings, and other metrics are recorded in Elasticsearch, enabling real-time monitoring of the infrastructure.

Kibana plays a key role in this monitoring process by allowing the creation of custom charts and visualizations. These visualizations enable the monitoring of various metrics related to the infrastructure. By leveraging Kibana's capabilities, we can build charts that provide insights into the performance and health of the infrastructure. These visualizations help in identifying any anomalies or errors that may occur during the execution of the plan.

Additionally, monitoring the execution of the plan allows us to compare the pre and post planned infrastructure states. This comparison helps identify any flaws or discrepancies that may have occurred during the plan execution. By detecting such errors, we can ensure that the plan is successfully applied and iterate the optimization process for the next cycle.

6 Evaluation & Results

We conducted evaluations on various use cases to assess the effectiveness of our system. These use cases primarily focused on optimizing costs, hard disk space utilization to avoid future expenses, minimizing latency between services, and maximizing throughput.

To set up the experiments, we created a rule index in Elasticsearch. This index served as a table where we added different rules corresponding to each use case. Our Cloud Simulator monitored this rule index to check for any additions or changes. Examples of sample rules are covered below.

Rule 1:

- **Use case:** Cost optimization
- **Status:** New
- **Rule:** ['vm_aws2', 'vm_aws3']
- **Description:** This rule simulates cloud cost optimization by adding a low load to the specified virtual machines, simulating idle behavior and enabling us to turn off these virtual machines when not needed.

Rule 2:

- **Use case:** Outbound optimization
- **Status:** New
- **Rule:** ['vm_azure3', 'vm_gcp3','vm_azure4']
- **Description:** This rule generates a high outbound load on the specified virtual machines. The generated load is later utilized by the Planner to enable caching and optimize outbound data transfer.

Rule 3:

- **Use case:** Hard disk optimization
- **Status:** New
- **Rule:** ['vm_azure4', 'vm_gcp4', 'db_aws1']
- **Description:** This rule generates a high hard disk load on the specified virtual machines. The simulator simulates the load, and the Planner checks if the hard disk is filled up to 80%. If the threshold is reached, the data is compressed, and the hard disk is cleaned to maintain 80% free space.

Additionally, we implemented other rules to deliberately change the region for different virtual machines that are dependent on databases. These experiments served as simple examples of the use cases we explored, but the actual usecases can be more complex and varied.

Through these experiments, we utilized our Planner to evaluate and optimize the cloud infrastructure based on the defined rules and usecase requirements. This approach allowed us to test the effectiveness of our system and assess its performance in real-world scenarios.

Provided below are some visual representations of each experiment and how infrastructure behaves after optimization is performed on it by our system. We will present a pre-plan and a post-plan infrastructure dashboard to showcase the transparency of our experiments and their results.

6.1 Experiments

Experiment 1: Cost Optimization

In this experiment, we simulate how the planner turns off virtual machines whose load is below a certain threshold, which in our case is 40%. All virtual machines specified in Rule 1 will be turned off. The cloud simulator adds low load to these virtual machines to simulate this experiment. In Figure 6.1, we observe that AWS currently has 6 virtual machines, resulting in a total cost of 3.024. The CPU load of AWS virtual machines 2 and 3 can be seen in Figure 6.2 and 6.3, respectively. We can observe that the CPU load of these virtual machines is decreasing due to the rule we added to reduce the load.

The learner module predicts the future state, indicating that these virtual machines will be turned off. This information is passed to the planner in the problem file, and the planner generates a shutdown plan for these virtual machines.

Plan Generated: `[('stop_application', 'vm_aws2'), ('turn_off', 'vm_aws2'), ('stop_application', 'vm_aws3'), ('turn_off', 'vm_aws3')]`

After applying the plan, we can observe in Figure 6.4 that now only 4 virtual machines are turned on instead of 6. As we are using a pay-as-you-go model, the cost is automatically reduced since the turned-off virtual machines are not accounted for in the bill.

Experiment 2: Hard Disk storage optimization

In Figure 6.5, we can observe that the hard disk load for Azure virtual machine 4 is above 80 percent. For brevity, we only include figures for virtual machine 4, but the behavior is the same for the remaining virtual machines defined in Rule 3.

After feeding the data to the planner, it generates the following plan:

Plan Generated: `[('compress_data', 'vm_azure4'), ('move_compress_data_to_cold_storage', 'vm_azure4'), ('clean_up_compress_data', 'vm_azure4'), ('compress_data', 'vm_gcp4'), ('move_compress_data_to_cold_storage', 'vm_gcp4'), ('clean_up_compress_data', 'vm_gcp4'), ('compress_data', 'db_aws1'), ('move_compress_data_to_cold_storage', 'db_aws1'), ('clean_up_compress_data', 'db_aws1')]`

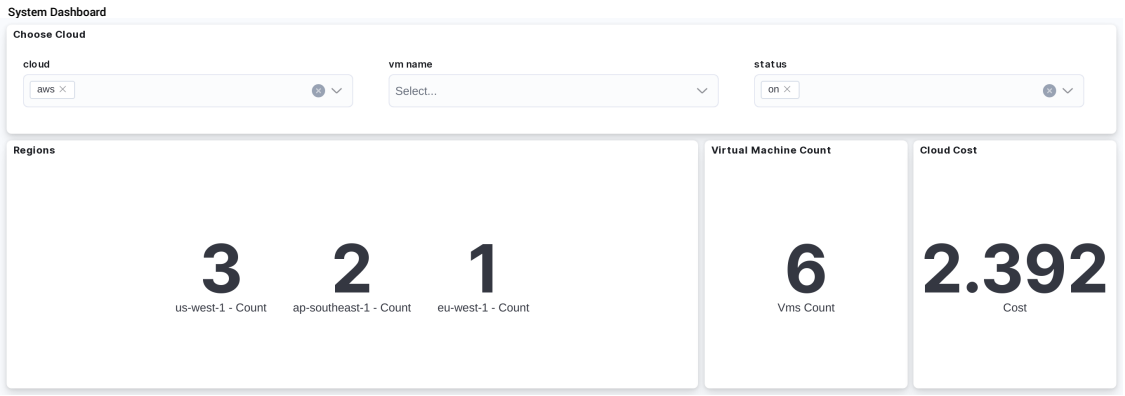

Figure 6.1: System overview before optimization

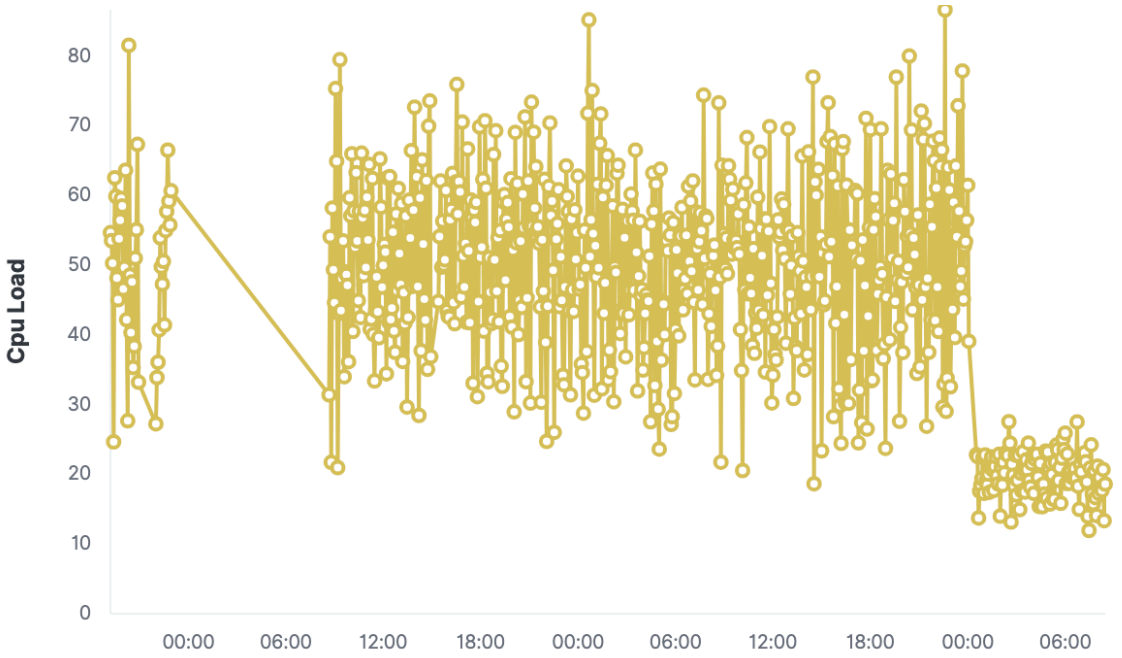

Figure 6.2: Load of AWS Virtual Machine 2

After executing the plan, all hard disk data is compressed and the filled storage is set to 20 percent. This can be observed in Figure 6.6 for Azure virtual machine 4.

Experiment 3: Reducing outbound traffic by enabling cache

In this experiment, we aim to demonstrate the cost savings achieved by enabling cache in virtual machines. Outbound traffic is typically costly in cloud environments, as it is often not unlimited.

Following Rule 2, our simulator generates high outbound traffic for Azure virtual machines 1 and 3, as well as GCP virtual machine 3. This traffic generation can be observed in in Figure 6.7, 6.8 and 6.9 respectively.

After providing this data to the planner, it generates the following plan:

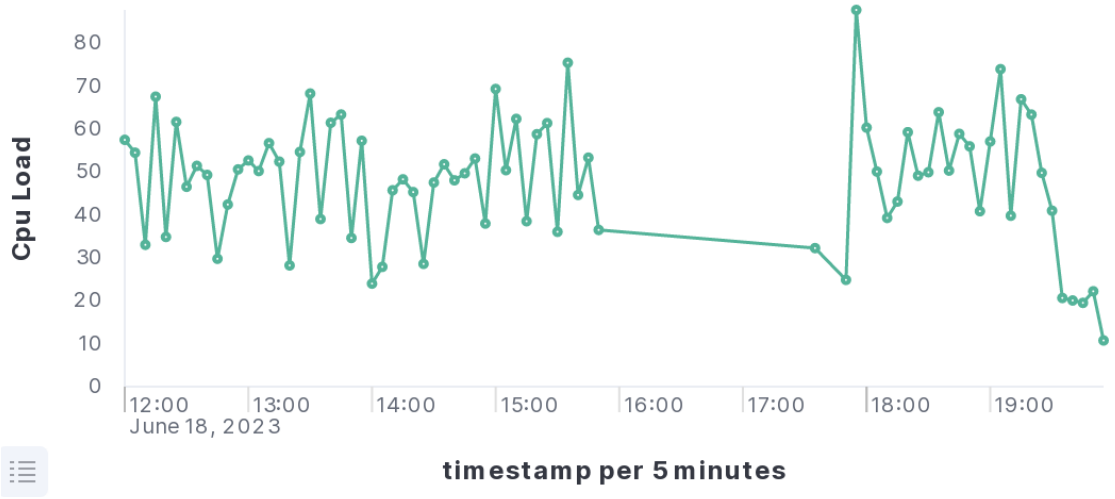

Figure 6.3: Load of AWS Virtual Machine 3

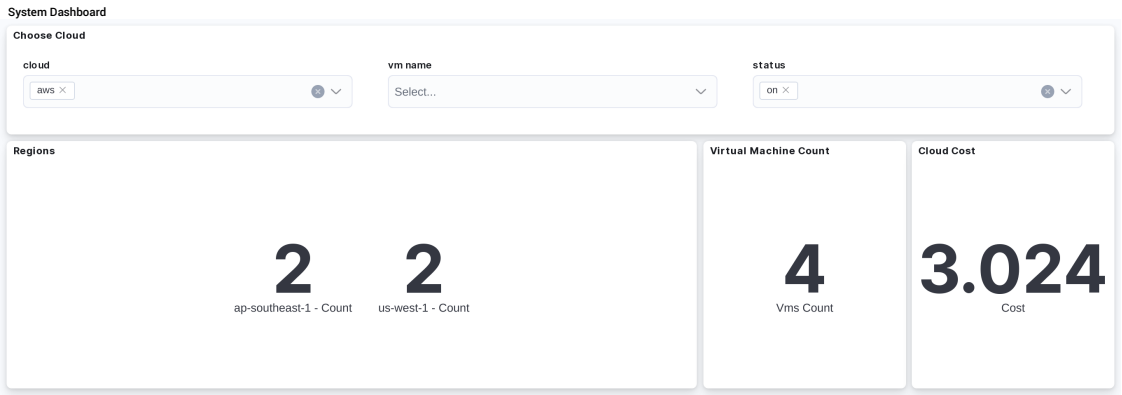

Figure 6.4: System overview after optimization

Plan Generated: `[('enable_cache', 'vm_azure1'), ('enable_cache', 'vm_azure3'),('enable_cache', 'vm_gcp3')]`

Upon executing the plan, we observe a significant reduction in outbound traffic for Azure virtual machines 1 and 3, as well as GCP virtual machine 3. This reduction can be visualized in Figure 6.10, 6.11 and 6.12, respectively.

Experiment 4: Virtual machine reconfiguration with Database dependencies

In our domain model, we consider the requirement that if a virtual machine is dependent on a database, both should be in the same region to avoid high throughput issues. In our infrastructure, we deliberately misconfigured the region of `vm_azure1`, which is dependent on the `db_azure1` database. The region of `vm_azure1` is set to "us-west-1", while the database is located in the `westeurope` region.

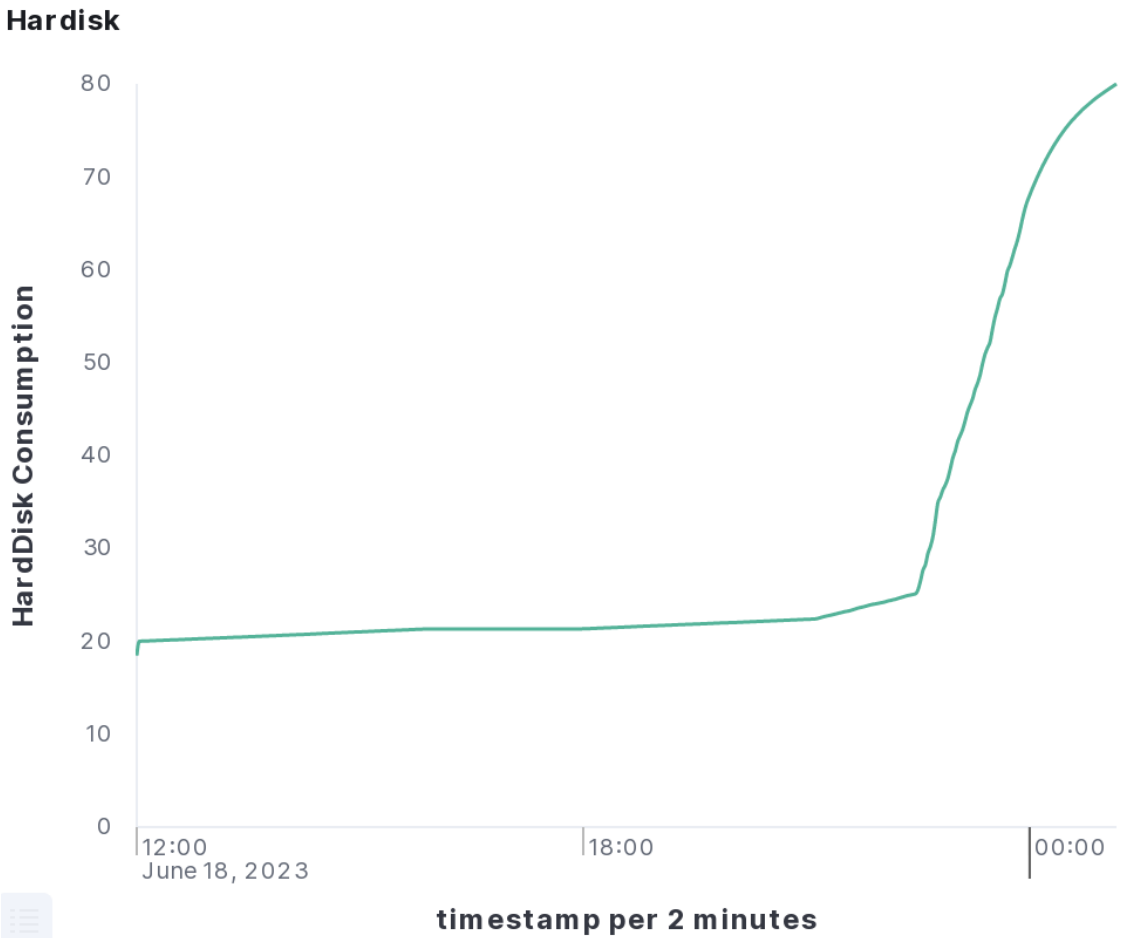

Figure 6.5: Hard Disk consumption of Azure VM 4 before optimization

We perform a query on Elasticsearch to change the region of `vm_azure1` to "us-east1" for our testcase. In 6.1, the Elasticsearch update query used is stated.

Before applying the plan, we can see the configuration of `vm_azure1` as stated in 6.2.

The plan generated to address the region mismatch is as follows:

Plan Generated: `[('turn_off', 'vm_azure1'), ('change_region', 'vm_azure1', 'westeurope'), ('turn_on', 'vm_azure1')]`

By executing this plan, we ensure that the region of `vm_azure1` is corrected to match the region of the `db_azure1` database, resolving the mismatch. This ensures that the virtual machine and database are in the same region, mitigating potential high throughput problems.

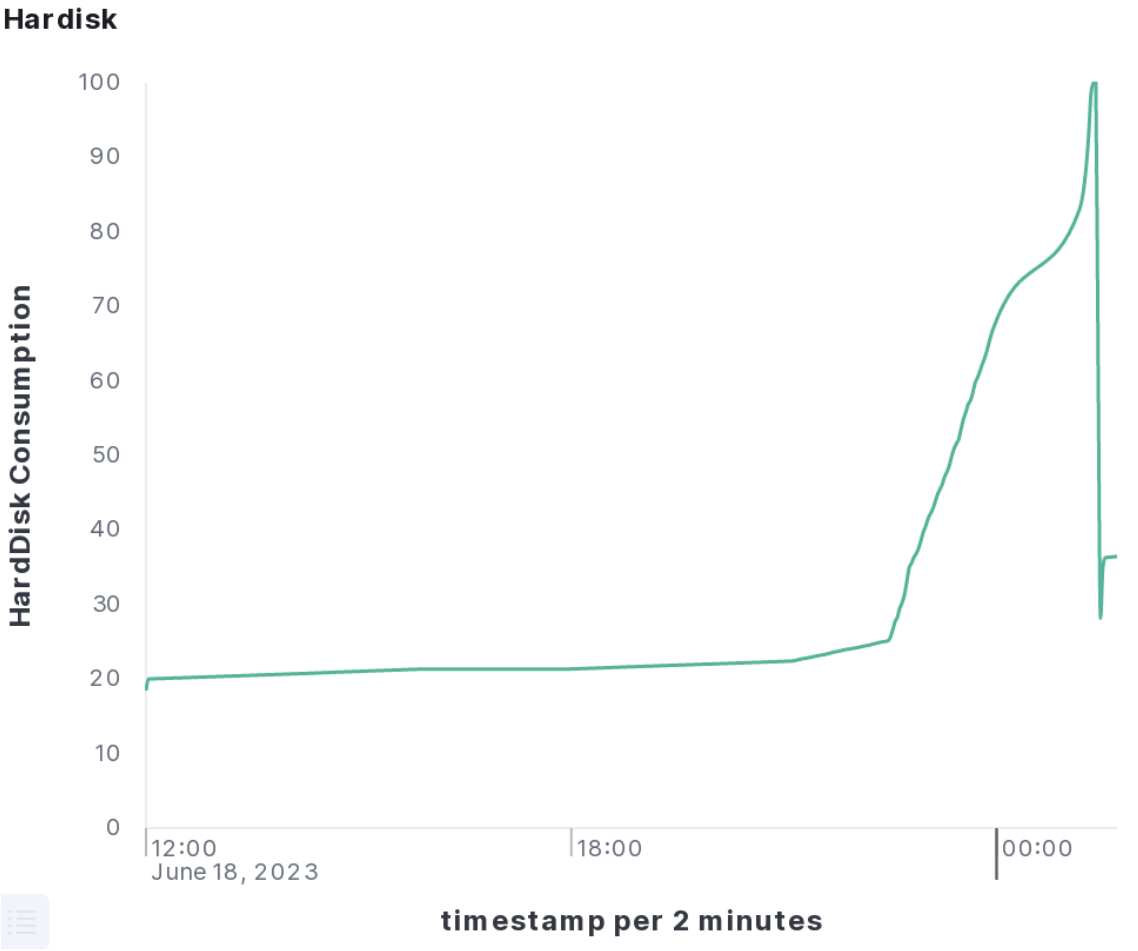

Figure 6.6: Hard Disk consumption of Azure VM 4 after optimization

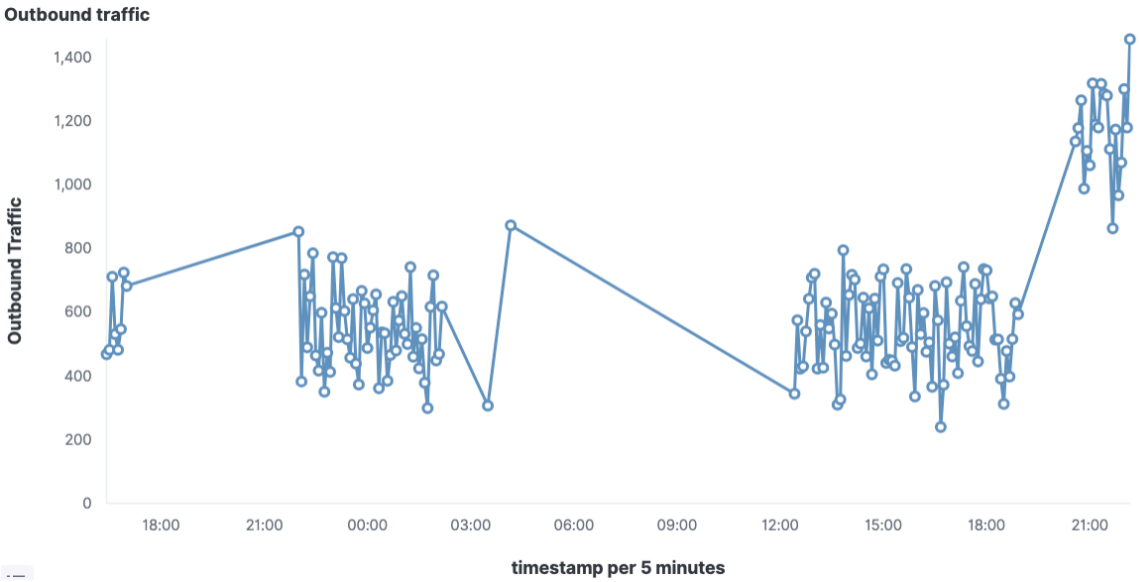

Figure 6.7: Outbound traffic of Azure VM 1 before optimization

Listing 6.1 Kibana query for changing region

```
POST vms_update_by_query
{
  "script": {
    "source": "ctx._source.region = params.region",
    "params": {
      "region": "us-east1"
    }
  },
  "query": {
    "term": {
      "name.keyword": {
        "value": "vm_azure1"
      }
    }
  }
}
```

Listing 6.2 Currently running Virtual Machines' configuration

```
{
  "type": "app",
  "dependencies": [
    "vm_azure2",
    "db_azure1"
  ],
  "vm_id": "0f4e5acd-bb06-4ed2-93ba-a7a1a1f7c2c9_vm_azure1",
  "instance_size": "Standard_B1s",
  "name": "vm_azure1",
  "caching_enabled": true,
  "region": "us-west-1",
  "cost-per-hour": 0.03,
  "application_installed": [
    "web",
    "python-microservice",
    "java-microservice"
  ],
  "status": "on"
}
```

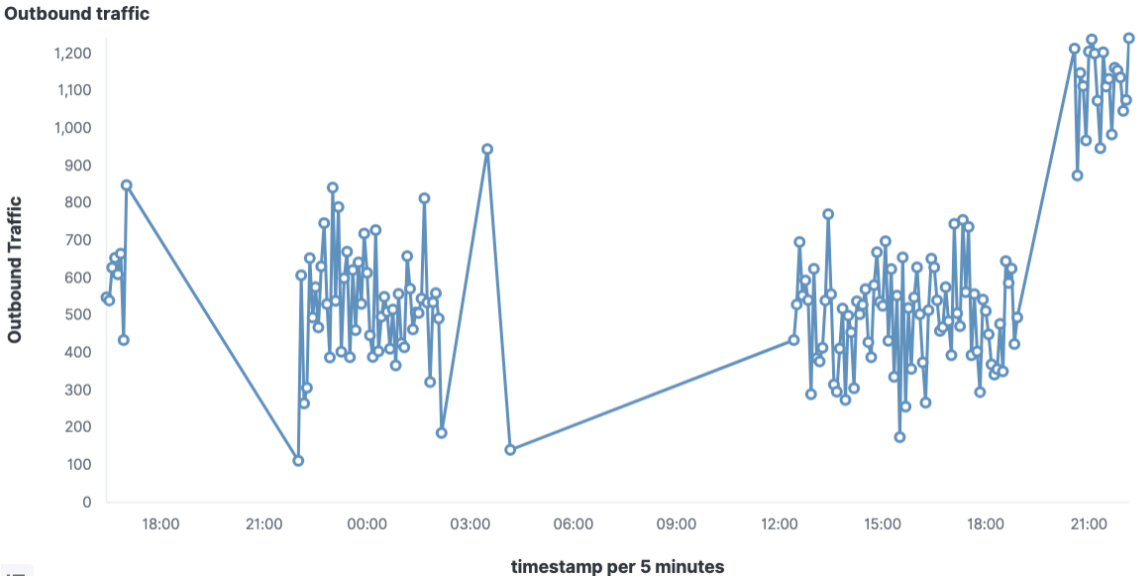

Figure 6.8: Outbound traffic of Azure VM 3 before optimization

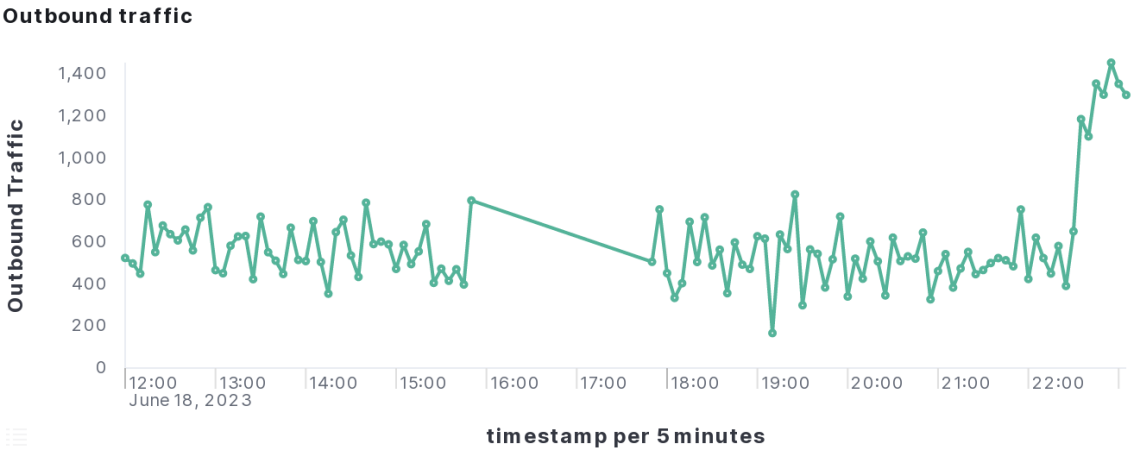

Figure 6.9: Outbound traffic of GCP VM 3 before optimization

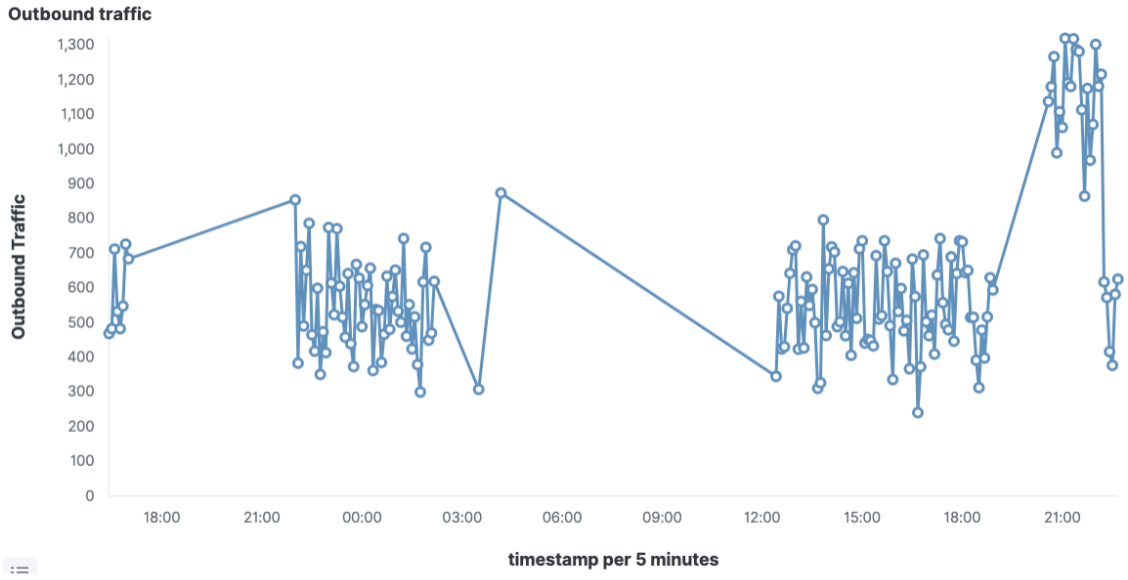

Figure 6.10: Outbound traffic of Azure VM 1 after optimization

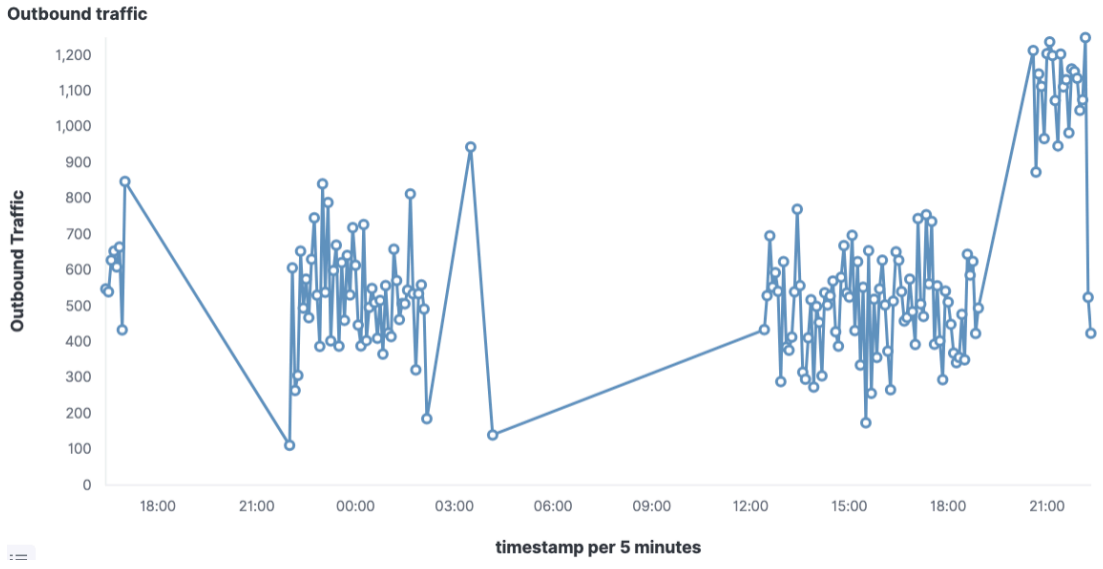

Figure 6.11: Outbound traffic of Azure VM 3 after optimization

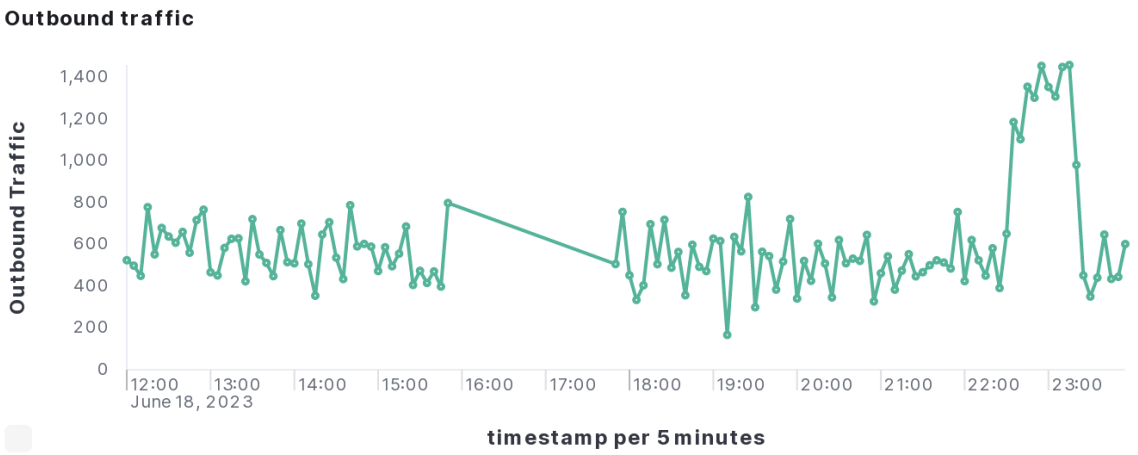

Figure 6.12: Outbound traffic of GCP VM 3 after optimization

7 Conclusion

In this thesis work, we have presented a system for optimizing cloud cost and infrastructure through automated planning. The system is designed to address the challenges of capturing real-time cloud data, generating problem files for Planners, and monitoring cloud infrastructure changes.

The system architecture consists of several components, including the Planner, Executor, Learner, Cloud Simulator, Elasticsearch, and Display. The Planner utilizes the GTPyhop HTN Planner to generate optimized plans based on the problem files generated from the Cloud Simulator and Learner module. The Executor executes these plans by communicating with the Cloud Simulator through API calls. The Learner module uses the FB Prophet time series library to predict future system load, which serves as a heuristic for identifying virtual machines that can be turned off to optimize cost.

To visualize metrics and results, we employ Kibana with Elasticsearch, allowing users to view and interact with the system's output and forecasts. Elasticsearch facilitates efficient storage and retrieval of large volumes of data, while Kibana provides a user-friendly interface for data visualization.

We have implemented the system by modeling the cloud infrastructure in the HTN domain and building a synthetic data generator in the Cloud Simulator. The Learner module predicts future system load based on historical data, and the Planner generates optimized plans considering the predicted load. The executor executes these plans using the Cloud Simulator and DSL query, and the results are displayed through the Kibana dashboard.

Through our system, we have demonstrated the effectiveness of automated planning in optimizing cloud cost and infrastructure. By utilizing the simulation environment, we can evaluate the performance of different optimization strategies and make informed decisions about turning off virtual machines. The integration of machine learning techniques enables accurate load prediction, further improving the optimization process. We test our system with different usecases as mentioned in evaluation section and all the time it correctly plan from given context of infrastructure.

Overall, our system provides a robust solution for optimizing cloud cost and infrastructure through automated planning. By leveraging the AI planning and timeseries forecasting, we are optimizing cloud resources and reducing costs while maintaining efficient performance. Our solution offers a flexible and scalable approach that can be extended to include additional optimization strategies and metrics.

Bibliography

[AHK+98] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. SRI, A. Barrett, D. Christianson, et al. "Pddl| the planning domain definition language". In: *Technical Report, Tech. Rep.* (1998) (cit. on p. 17).

[ANE13] S. Abrishami, M. Naghibzadeh, D. H. Epema. "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds". In: *Future generation computer systems* 29.1 (2013), pp. 158–169 (cit. on p. 25).

[BAH19] P. Bercher, R. Alford, D. Höller. "A Survey on Hierarchical Planning-One Abstract Idea, Many Concrete Realizations." In: *IJCAI*. 2019, pp. 6267–6275 (cit. on pp. 19, 20).

[ENS95] K. Erol, D. S. Nau, V. S. Subrahmanian. "Complexity, decidability and undecidability results for domain-independent planning". In: *Artificial intelligence* 76.1-2 (1995), pp. 75–88 (cit. on p. 18).

[GA14] I. Georgievski, M. Aiello. "An overview of hierarchical task network planning". In: *arXiv preprint arXiv:1403.7426* (2014) (cit. on pp. 19, 20).

[GNLA17] I. Georgievski, F. Nizamic, A. Lazovik, M. Aiello. "Cloud ready applications composed via HTN planning". In: *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2017, pp. 81–89 (cit. on p. 25).

[GNT04] M. Ghallab, D. Nau, P. Traverso. *Automated Planning: theory and practice*. Elsevier, 2004 (cit. on p. 17).

[HBBB21] D. Höller, G. Behnke, P. Bercher, S. Biundo. "The PANDA framework for hierarchical planning". In: *KI-Künstliche Intelligenz* (2021), pp. 1–6 (cit. on p. 19).

[HLMM19] P. Haslum, N. Lipovetzky, D. Magazzeni, C. Muise. "An introduction to the planning domain definition language". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 13.2 (2019), pp. 1–187 (cit. on p. 17).

[KS+92] H. A. Kautz, B. Selman, et al. "Planning as Satisfiability." In: *ECAI*. Vol. 92. Citeseer. 1992, pp. 359–363 (cit. on p. 18).

[MH11] M. Mao, M. Humphrey. "Auto-scaling to minimize cost and meet application deadlines in cloud workflows". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12 (cit. on p. 25).

[NBP+21] D. Nau, Y. Bansod, S. Patra, M. Roberts, R. Li. "GTPyhop: A hierarchical goal+ task planner implemented in Python". In: *HPlan 2021* (2021), p. 21 (cit. on pp. 21, 23).

[ON20] P. Osypanka, P. Nawrocki. "Resource usage cost optimization in cloud computing using machine learning". In: *IEEE Transactions on Cloud Computing* 10.3 (2020), pp. 2079–2089 (cit. on p. 25).

[Sac75] E. D. Sacerdoti. *The nonlinear nature of plans*. Tech. rep. STANFORD RESEARCH INST MENLO PARK CA, 1975 (cit. on p. 19).

[SAGS14] Y. Siahmargooei, M. K. Akbari, S. A. H. Golpayegani, S. Sharifian. "Near-Optimal Virtual Machine Packing Based on Resource Requirement of Service Demands Using Pattern Clustering". In: *arXiv preprint arXiv:1406.7285* (2014) (cit. on p. 25).

[SDN14] S. Sowmya, P. Deepika, J. Naren. "Layers of cloud–IaaS, PaaS and SaaS: a survey". In: *International Journal of Computer Science and Information Technologies* 5.3 (2014), pp. 4477–4480 (cit. on p. 13).

[WC15] E. Weintraub, Y. Cohen. "Cost optimization of cloud computing services in a networked environment". In: *International Journal of Advanced Computer Science and Applications* 6.4 (2015) (cit. on p. 25).

[Wil14] D. E. Wilkins. *Practical planning: extending the classical AI planning paradigm*. Elsevier, 2014 (cit. on p. 19).

[Yan90] Q. Yang. "Formalizing planning knowledge for hierarchical planning". In: *Computational intelligence* 6.1 (1990), pp. 12–24 (cit. on p. 19).

[YXJ+18] J. Yang, W. Xiao, C. Jiang, M. S. Hossain, G. Muhammad, S. U. Amin. "Ai-powered green cloud and data center". In: *IEEE Access* 7 (2018), pp. 4195–4203 (cit. on p. 25).

[ZCY+10] G. Zou, Y. Chen, Y. Yang, R. Huang, Y. Xu. "AI planning and combinatorial optimization for web service composition in cloud computing". In: *Annual International Conference on Cloud Computing and Virtualization (CCV 2010)*. 2010, pp. 1–8 (cit. on p. 25).

All links were last followed on June 21, 2023.

A Appendix

```
def turn_off(state, vm):
    """
    Turns off the specified virtual machine if it is currently turned on.
    """
    if state.vms[vm]['status'] == "on":
        state.vms[vm]['status'] = "off"
        return state

def turn_on(state, vm):
    """
    Turns on the specified virtual machine if it is currently turned off.
    """
    if state.vms[vm]['status'] == "off":
        state.vms[vm]['status'] = "on"
        return state

def clean_up_compress_data(state, vm):
    """
    Sets the 'signal_clean_up_compress_data' flag to True for the specified virtual
machine.
    """
    if not state.vms[vm]['signal_clean_up_compress_data']:
        state.vms[vm]['signal_clean_up_compress_data'] = True
        return state

def move_compress_data_to_cold_storag(state, vm):
    """
    Sets the 'signal_move_compress_data_to_cold_storage' flag to True for the
specified virtual machine.
    """
    if not state.vms[vm]['signal_move_compress_data_to_cold_storag']:
        state.vms[vm]['signal_move_compress_data_to_cold_storag'] = True
        return state

def compress_data(state, vm):
    """
    Sets the 'signal_compress_data' flag to True for the specified virtual machine.
    """
```

```
    if not state.vms[vm]['signal_compress_data']:
        state.vms[vm]['signal_compress_data'] = True
        return state

def create_snapshop(state, vm):
    """
    Sets the 'signal_create_snapshot' flag to True for the specified virtual machine.
    """
    if not state.vms[vm]['signal_create_snapshop']:
        state.vms[vm]['signal_create_snapshop'] = True
        return state

def stop_application(state, vm,app=None):
    """
    Sets the 'signal_stop_application' flag to True for the specified application
running on the virtual machine.
    """
    if not state.vms[vm]['signal_stop_application']:
        state.vms[vm]['stop_application'] = True
        return state

def enable_cache(state, vm):
    """
    Enables caching for the specified virtual machine if it is currently disabled.
    """
    if state.vms[vm]['caching_enabled'] == False:
        state.vms[vm]['caching_enabled'] = True
        return state

def scale_down_app(state,vm):
    """
    Reduces the number of application instances running on the virtual machine if
there is no traffic.
    """
    if state.vm[vm]['replication_factor'] > 1:
        state.vm[vm]['replication_factor'] -= 1
        return state

def stop_applications(state, vm):
    """
    Stops all applications running on the specified virtual machine.
    """
    if len(state.vms[vm]['application_installed'])>0:
        return [("stop_application",vm)]
    return []

def deallocate_virtual_machine(state, vm):
```

```
    """
    Deallocates the specified virtual machine by stopping its applications and turning
 it off.
    """

    return [("stop_applications",vm),("turn_off",vm)]#actions

def optimized_storage_cost(state, vm):
    """
    Performs actions to optimize storage cost for the specified virtual machine if
certain conditions are met.
    """
    if  np.mean((state.vms[vm]['avg_hardisk'])) >= 80:
        return [("compress_data",vm),("move_compress_data_to_cold_storag",vm),("
clean_up_compress_data",vm)]
    else:
        return []

def optimized_network_cost(state, vm):
    """
    Performs actions to optimize network cost for the specified virtual machine if
certain conditions are met.
    """
    if np.mean(state.vms[vm]['avg_outbound']) > 800 and not (state.vms[vm]['
caching_enabled']):
        return [("enable_cache",vm)]
    else:
        return []

def change_region(state, vm, new_region):
    """
    Changes the region of the specified virtual machine to the new region.
    """
    if state.vms[vm]["region"] != new_region and state.vms[vm]['status'] == "off":
        state.vms[vm]["region"] = new_region
        return state

def is_vm_dependent_on_database(vm_data):
    """
    Checks if the virtual machine has dependencies on a database.
    """
    for dependency in vm_data['dependencies']:
        if dependency.startswith("db_"):
            return True
    return False

def get_dep_db( vm_data):
```

```
    """
    Returns the name of the database dependency for the virtual machine.
    """
    for dependency in vm_data['dependencies']:
        if dependency.startswith("db_"):
            return dependency
    return None

def check_region(vm1, db):
    """
    Checks if the region of the virtual machine matches the region of the database.
    """
    if vm1["region"] == db["region"]:
        return True
    return False

def scale_down(state, vm):
    """
    Scales down the specified virtual machine by reducing the replication factor to 1.
    """
    if state.vms[vm]["replication_factor"] > 1:
        state.vms[vm]["replication_factor"] -= state.vms[vm]["replication_factor"]
        return state

def re_scale_replicas(state, vm):
    """
    Rescales the replicas of the virtual machine if certain conditions are met.
    """
    if state.vms[vm]['type'] == "db" and np.mean(state.vms[vm]['avg_cpu'])<40 and
state.vms[vm]["future_status"] == "on" and state.vms[vm]["replication_factor"] > 1:#
and np.mean(state.vms[vm]["avg_cpu"]) < 60:
        return [("scale_down", vm)]
    return []

def re_configure_vm(state, vm):
    """
    Reconfigures the virtual machine if certain conditions are met.
    """
    if state.vms[vm]["status"] != "off":
        if is_vm_dependent_on_database(state.vms[vm]):
            db_name = get_dep_db(state.vms[vm])
            # stop expansion
            if db_name is None:
                return []

            if  state.vms[vm]['region'] != state.vms[db_name]['region']:
```

```
                return [("turn_off", vm),("change_region",vm,state.vms[db_name]['
region']),("turn_on",vm)]
    return []

def optimize_cloud_costs(state, vm):
    """
    Optimizes cloud costs for the specified virtual machine by performing various
actions.
    """
    if state.vms[vm]['status'] == "on" and state.vms[vm]['future_status'] == "on":
        return [('optimized_network_cost', vm),("optimized_storage_cost",vm),("
re_configure_vm",vm),("re_scale_replicas",vm)]
    else:
        return [("deallocate_virtual_machine",vm)]

# define domain name which is file
gtpyhop.current_domain = the_domain

# define all actions here
gtpyhop.declare_operators(turn_off,turn_on,scale_down,change_region,enable_cache,
compress_data,move_compress_data_to_cold_storag,clean_up_compress_data,create_snapshop
,stop_application)

# define all methods here
gtpyhop.declare_methods('optimize_cloud_costs', optimize_cloud_costs)
gtpyhop.declare_methods('optimized_network_cost', optimized_network_cost)
gtpyhop.declare_methods('optimized_storage_cost', optimized_storage_cost)
gtpyhop.declare_methods('deallocate_virtual_machine', deallocate_virtual_machine)
gtpyhop.declare_methods('re_configure_vm', re_configure_vm)
gtpyhop.declare_methods('re_scale_replicas', re_scale_replicas)
gtpyhop.declare_methods('stop_applications', stop_applications)
gtpyhop.verbose = 2

def plan(data):
    state.vms = copy.deepcopy(data)
    solution = gtpyhop.find_plan(state, [('optimize_cloud_costs', vm) for vm in state.
vms.keys()])
    if not solution:
        print("No solution found")
        return None
    return solution
```

Listing A.1: Domain file in GTPyhop

```
        {
   "vm_aws1":{
      "signal_create_snapshop":true,
```

```
"signal_stop_application":false,
"signal_move_compress_data_to_cold_storag":true,
"signal_compress_data":true,
"signal_clean_up_compress_data":true,
"schedule_task":[

],
"type":"app",
"application_installed":[
   "nginx"
],
"cloud":"aws",
"load":{
   "nginx":0.2,
   "sql":0.4
},
"status":"on",
"instance_size":"t2.micro",
"cost-per-hour":0.02,
"dependencies":[
   "vm_aws2",
   "db_aws1"
],
"hard_disk_consumption":50,
"caching_enabled":true,
"region":"us-west-1",
"name":"vm_aws1",
"vm_id":"86ff8a20-2ef4-4010-92f4-e6021ead1385_vm_aws1",
"avg_ram":[
   61.17048263549805,
   56.367958068847656
],
"avg_hardisk":[
   96.0,
   96.0
],
"avg_latency_0":[
   285.5109558105469,
   478.27545166015625
],
"avg_latency_1":[
   109.38603210449219,
   451.8224792480469
],
"avg_load_0":[
   77.77836608886719,
   21.198774337768555
```

```
      ],
      "avg_cpu":[
         79.53987121582031,
         11.0060453414917
      ],
      "avg_outbound":[
         458.2621154785156,
         723.30517578125
      ],
      "future_status":"on",
      "stop_application":true
   }
}
```

Listing A.2: Problem file in GTPyhop for single virtual machine

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature