UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Software Engineering Curriculum

Volodymyr Chernetskyi

Container-Based Microservice Placement Optimization in Cloud

Master's Thesis (30 ECTS)

Supervisor: Chinmaya Kumar Dehury, PhD

Tartu 2023

Container-Based Microservice Placement Optimization in Cloud

Abstract:

Today, many applications operate as containerized microservices running on multi-machine clusters in cloud computing environments. High container-machine placement quality increases throughput, reduces latency, and improves cluster resource utilization. Optimal placement is achieved by solving complex multidimensional optimization problems. However, in response to the highly volatile nature of the cloud environment, modern container management systems opt for suboptimal placement, prioritizing quick decision-making. Over time, these poor decisions compound, increasing the need for placement reevaluation. Available reevaluation solutions change the placement of containers by making them undergo the same placement process again, with no guarantee of improved results. To address the issue, this thesis presents a novel placement reevaluation algorithm based on the particle swarm optimization approach with a focus on optimizing infrastructure costs. By integrating interzonal network traffic costs, the proposed algorithm achieves cost reductions beyond those obtained through conventional solutions that focus solely on computing resource costs. The stability of the algorithm in high-dimensional tightly bounded discrete search spaces was enhanced through improved particle position initialization. Empirical evaluations demonstrate the efficiency of the proposed algorithm, surpassing traditional optimization problem solvers, while outperforming standard particle swarm optimization implementations in terms of accuracy.

Keywords:

Particle swarm optimization, Kubernetes, offline scheduling, microservice architecture, cost optimization, cloud computing, distributed systems.

CERCS: P170 Computer science, numerical analysis, systems, control

Konteinerpõhine mikroteenuste paigutuse optimeerimine pilves

Lühikokkuvõte:

Tänapäeval töötavad paljud rakendused konteinerite mikroteenustena, mis töötavad mitme masinaga klastrites pilvandmetöötluse keskkondades. Konteinerite ja masinate paigutuse kõrge kvaliteet suurendab läbilaskevõimet, vähendab latentsust ja parandab klastri ressursside kasutamist. Optimaalne paigutus saavutatakse keeruliste mitmemõõtmeliste optimeerimisülesannete lahendamisega. Kuid vastuseks pilvekeskkonna väga muutlikule olemusele valivad kaasaegsed konteinerihaldussüsteemid mitteoptimaalse paigutuse, eelistades kiiret otsustamist. Aja jooksul need kehvad otsused süvenevad, suurendades vajadust paigutuse ümberhindamise järele. Saadaolevad ümberhindamise lahendused muudavad konteinerite paigutust, pannes need uuesti läbi sama paigutusprotsessi, ilma et tulemused paraneksid. Selle probleemi lahendamiseks esitatakse selles lõputöös uudne paigutuse ümberhindamise algoritm, mis põhineb osakeste sülemi optimeerimise lähenemisviisil, keskendudes infrastruktuuri kulude optimeerimisele. Integreerides tsoonidevahelisi võrguliikluskulusid, saavutab pakutud algoritm kulude vähendamise, mis on suurem kui tavapäraste lahenduste abil, mis keskenduvad ainult ressursikulude arvutamisele. Algoritmi stabiilsust suuremõõtmelistes tihedalt piiratud diskreetsetes otsinguruumides suurendas osakeste positsiooni täiustatud lähtestamine. Empiirilised hinnangud näitavad pakutud algoritmi tõhusust, ületades traditsioonilisi optimeerimisprobleemide lahendajaid, ületades samal ajal täpsuse osas standardseid osakeste sülemi optimeerimise rakendusi.

Võtmesõnad:

Osakeste sülemi optimeerimine, Kubernetes, võrguühenduseta ajakava, mikroteenuste arhitektuur, kulude optimeerimine, pilvandmetöötlus, hajutatud süsteemid.

CERCS: P170 Arvutiteadus, arvanalüüs, süsteemid, kontroll

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ACO	Ant Colony Optimization
AKS	Azure Kubernetes Service
API	application programming interface
AWS	Amazon Web Services
AZ	availability zone
CaaS	Container as a Service
CapEx	capital expenditure
CP-SAT	constraint programming-boolean satisfiability
CPU	central processing unit
CSP	cloud service provider
DC	data center
DNS	Domain Name System
EKS	Elastic Kubernetes Service
FaaS	Function as a Service
GCP	Google Cloud Provider
GKE	Google Kubernetes Engine
GPU	graphics processing unit
IaaS	Infrastructure as a Service
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
NP	nondeterministic polynomial
OpEx	operating expense
OS	operating system
PaaS	Platform as a Service
PSO	Particle Swarm Optimization
QoS	Quality of Service
RAM	random access memory
SaaS	Software as a Service
SAT	boolean satisfiability
USD	United States dollar
VM	virtual machine
VMM	virtual machine monitor

1 Introduction

In recent years, microservices have become increasingly popular. Many organizations have adopted this architectural style for software development [1]. The popularity of microservices can be attributed to several benefits, including improved scalability, agility, and resilience. Microservices enable organizations to break down large monolithic applications into smaller and more manageable services that can be developed, deployed, and maintained independently. This approach allows for more frequent releases and a faster time-to-market.

One way to operate microservices is to use containers. Containers provide a lightweight, isolated runtime environment that enables multiple services to run on a single host, thereby improving resource utilization and reducing costs. Containers enable developers to package and deploy microservices as independent units, along with their dependencies and configurations, making moving them between different platforms easy.

Cloud computing has grown in popularity, with organizations of all sizes moving their applications to the cloud [2]. Organizations leverage cloud computing to shift from capital expenditures (CapEx) on building their own data centers (DCs) to operating expenses (CapEx) on used infrastructure, which is cheaper because of the higher economies of scale. The cloud also allows going beyond the planned infrastructure or data centers in minutes to provide customers the necessary agility.

One crucial aspect of running containerized microservice applications in the cloud is the placement of containers on machines. Placement is influenced by the requirements to the application or machines. One machine may be chosen over the other because of the minimum required computing resources or specific hardware, such as GPUs, which are in high demand owing to recent advancements in artificial intelligence. Organizations must carefully consider these factors when deciding how to place containers because poor placement decisions can lead to reduced performance, increased costs, and application unavailability.

Managing a large number of containers in a distributed system is challenging. Container orchestration is the process of automating the management, scaling, and deployment of containerized applications. Kubernetes[1] is widely recognized as the most popular system among various container orchestration platforms [3][4]. Kubernetes employs an online scheduling model to manage newly created or scaled-out containerized workloads, processing their requirements individually to select the most suitable machine available. However, this placement process lacks a global perspective. It does not consider other workloads waiting for placement or the state of the system after making a placement decision. These locally optimal decisions can lead to globally suboptimal outcomes, such as under- or overutilized machines, the "noisy neighbors" problem, network throttling, or single points of failure.

[1] `https://kubernetes.io/`

According to research findings [5], most enterprises aim to allocate a significant portion of their IT hosting budget, specifically 8 out of every 10 USD, to cloud services by 2024. Moreover, a separate study [6] highlighted that compute services constitute at least a quarter of the total cloud expenditure. Consequently, even incremental improvements such as optimizing the placement of microservices within cloud environments have the potential to yield substantial reductions in cloud-related costs.

While external tools are available to address the outcomes of suboptimal placement, they do not address the root cause, the online scheduling model. Descheduler[2] is a community-maintained project that addresses the high volatility of the Kubernetes cluster by evicting running workloads where possible for them to be rescheduled, hopefully optimizing placement. Karpenter[3] is an open-source Kubernetes cluster autoscaler built by Amazon Web Services (AWS) that can deprovision machines entirely or replace them with cheaper variants when placing their workloads differently is possible. Both tools rely on Kubernetes to schedule evicted workloads; however, there is no guarantee that the placement decision will improve next time.

This thesis focuses on providing an optimal container-machine placement for microservice applications. This thesis uses Kubernetes as an example, since it is the most popular orchestrator in the industry. Kubernetes provides a set of abstractions helpful in discussing the problem. However, the proposed solution is not limited to Kubernetes. It can be applied to any similar distributed system that runs containerized microservices. The solution leverages the benefits of the cloud environment, where Kubernetes often operates, specifically the wide variety of compute instance types, and the ability to scale on demand.

1.1 Motivation

The motivation behind this thesis stems from the recognition of a significant gap in the market: a lack of available tools capable of suggesting new placement schemes that effectively increase resource utilization while simultaneously decreasing infrastructure costs. The absence of a comprehensive solution has created a compelling need for novel approaches that address the complexities of microservice placement optimization in cloud computing environments.

1.2 Goals

The primary objective of this thesis is to design and implement an advanced placement reevaluation algorithm that demonstrates high accuracy without compromising the speed. The algorithm should incorporate various real-world cloud cost factors to effectively

[2] `https://sigs.k8s.io/descheduler`
[3] `https://karpenter.sh/`

reduce cloud costs. Through the successful implementation of such an innovative algorithm, this thesis aims to offer a valuable tool for enterprises seeking to decrease infrastructure expenses by optimizing their microservice placement.

1.3 Contributions

This thesis makes the following contributions:

- Catered solution to cloud customers, emphasizing a reduction in cloud infrastructure costs rather than prioritizing energy efficiency, hardware longevity, and tenant Quality of Service (QoS), which are commonly emphasized by cloud providers.
- Included network data transfer costs in the system model increasing the granularity of the objective function.
- Highlighted limitations of applying the widely used Particle Swarm Optimization (PSO) algorithm to the problem.
- Enhanced implementation of the PSO algorithm by incorporating a first-fit bin packing algorithm to improve the stability of the algorithm, fostering more consistent and reliable performance throughout its iterations.
- Conducted a comparative analysis between the proposed PSO-based solution and the standard PSO and linear solver through a series of experiments.

1.4 Thesis Outline

Chapter 2 establishes terminology and surveys related research, setting the context for the thesis. Chapter 3 defines the system model, including cloud components, application elements, assumptions, and notations. Chapter 4 presents the proposed algorithm's implementation, detailing design choices and rationales. Chapter 5 outlines experiments, covering setup, metrics, methodologies, and results. Chapter 6 concludes findings and suggests future research directions.

2 State of the Art

This chapter provides an in-depth explanation of the terminologies employed throughout the remainder of this thesis. A comprehensive analysis of related work is presented, offering insights into prior research and establishing the context for the ensuing discussions.

2.1 Background

2.1.1 Cloud Computing

Cloud computing is a model for enabling ubiquitous, convenient, and on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [7]. A company or organization that offers cloud computing resources is called a cloud service provider (CSP). Cloud service providers vary in size and services they offer. Amazon Web Services[4], Microsoft Azure[5], and Google Cloud Provider (GCP)[6] are the most common cloud service providers.

Cloud infrastructure (hereafter, infrastructure) is a collection of hardware and software that enables the essential characteristics of cloud computing. The cloud infrastructure can be viewed as containing both a physical layer and an abstraction layer. The physical layer comprises the hardware resources necessary to support the services; it typically includes servers, storage, and network components. The abstraction layer consists of the software deployed across the physical layer, which manifests essential cloud characteristics. Conceptually the abstraction layer sits above the physical layer [7].

Infrastructure is housed on special premises called data centers. An availability zone (AZ) is one or more closely interconnected data centers. Availability zones are isolated - the unavailability of one does not imply the unavailability of the other. The availability zones are grouped into regions that may further form the geographies. Azure's geography *Europe* consists of two regions: *North Europe* in Ireland and *West Europe* in the Netherlands; each of the regions consists of three availability zones. The AWS spans 99 availability zones within 31 regions, with at least three availability zones in each region [8].

As cloud computing has grown in popularity, several models and deployment strategies have emerged to help meet specific needs of different users: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS). Infrastructure as a Service is the capability provided to the consumer to provision processing, storage, networks, and other fundamental computing resources where the consumer can deploy

[4] `https://aws.amazon.com/`
[5] `https://azure.microsoft.com/`
[6] `https://cloud.google.com/`

and run arbitrary software. Platform as a Service is the capability provided to the consumer to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure. Software as a Service is the capability provided to the consumer to use the provider's applications running on a cloud infrastructure. [7]. Figure 1 explains how the responsibility is split between the cloud provider and the customer in these cloud models.

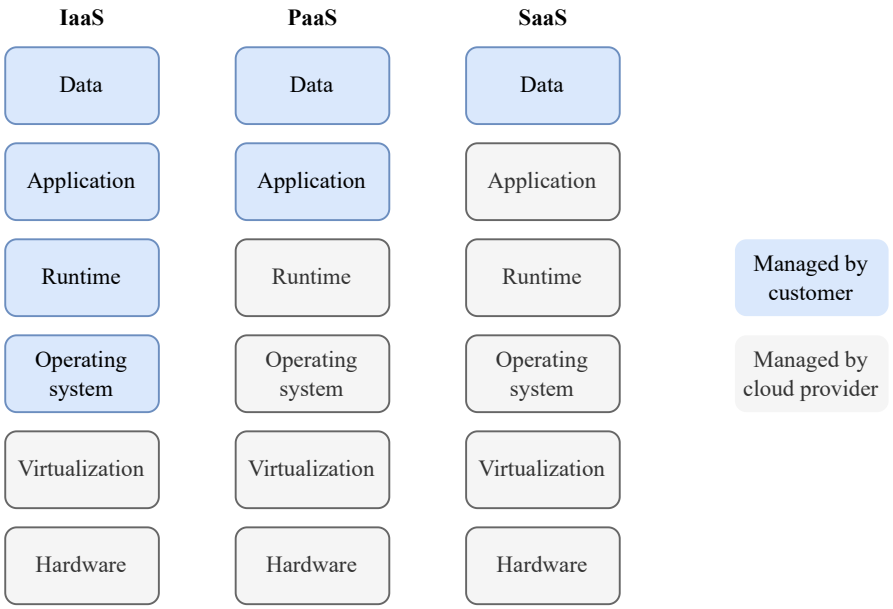

Figure 1. Comparison of IaaS, PaaS, and SaaS cloud models.

2.1.2 Virtualization and Containerization

Cloud infrastructure relies heavily on virtualization, that is, the simulation of the software and hardware upon which other software runs. This simulated environment is called a virtual machine (VM). Virtualization is enabled by a hypervisor (also known as a virtual machine monitor (VMM)). A hypervisor manages the guest operating systems (OSs) on the host and controls the instruction flow between them and the physical hardware [9]. There are two types of hypervisors that differ in their architecture and performance. The type 1 hypervisor (bare-metal hypervisor) is installed directly on the hardware. The type 2 hypervisor runs as software on a host OS. Despite their differences, both types of hypervisors have their use. Because of their low resource footprint, type 1 hypervisors are used in cloud data centers. Type 2 hypervisors are popular among consumers because

they are more user-friendly [10]. Examples of hypervisors include Kernel-based Virtual Machine (KVM)[7], Hyper-V[8], and VMware vSphere[9].

While VMs provide the isolation of operating systems, they may be too heavy for application isolation. Containers are lightweight alternatives to VMs. A container is a method for packaging and securely running an application in a virtualization environment [11]. The container engine is responsible for managing the container lifecycle. Common container engines include Linux Containers (LXC)[10], Docker[11], and Podman[12].

Figure 2 shows the different abstraction layers on which the virtual machines and containers are built. Although a hypervisor is not required for running containers, it is always present in cloud environments to virtualize operating systems for multiple tenants sharing the same hardware [10]. The critical difference between the VM and container is that the former ships with its own OS, whereas the latter utilizes host OS kernel features to run as an isolated process. The absence of an operating system inside the container is the reason for its reduced size. A container engine is required to manage the container lifecycle; however, running containers relies only on the OS kernel features. The container engine usually runs in userspace, thus its failure does not affect the running containers.

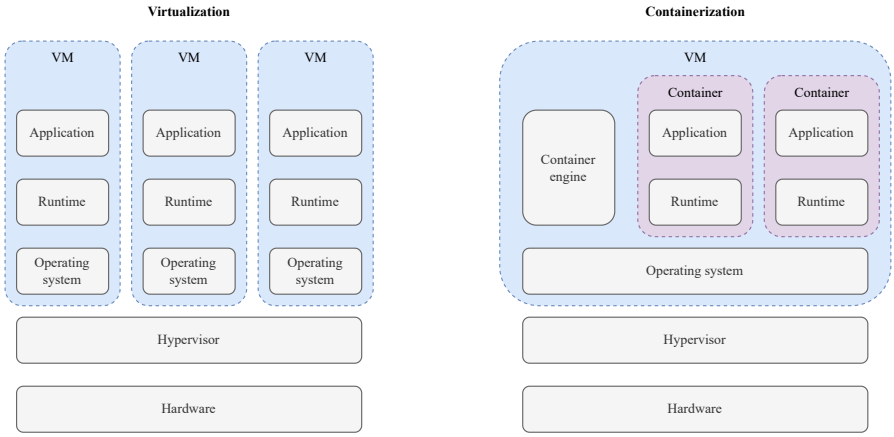

Figure 2. Comparison of virtual machine and container models.

New technologies such as Firecracker microVMs[13] represent an upgrade over traditional containers. MicroVMs are based on a minimalist kernel to reduce the memory

[7] https://www.linux-kvm.org/page/Main_Page
[8] https://learn.microsoft.com/virtualization/hyper-v-on-windows/about/
[9] https://www.vmware.com/products/vsphere.html
[10] https://linuxcontainers.org/
[11] https://www.docker.com/
[12] https://podman.io/
[13] https://firecracker-microvm.github.io/

footprint and attack surface area. Characterized by VM-like security and container-like startup times, microVMs are used in Container as a Service (CaaS) and Function as a Service (FaaS) offerings. Although this thesis relies on the word "container", the system may use microVMs without loss of generality.

2.1.3 Microservice Architecture

The microservice architectural style for developing software applications emphasizes breaking down the system into loosely coupled and independently deployable services. Unlike its counterpart, the traditional monolithic architecture, in which the entire system is built as a single unit, the microservice architecture allows developers to build and maintain the application as a collection of smaller interconnected services. Each service is designed to perform a specific business function, and can be developed, tested, and deployed independently. This approach enables faster development cycles because services can be updated and released without disrupting the entire system. It also improves scalability and resilience because individual services can be scaled depending on demand, and the failure of one service does not affect the entire system [12].

Figure 3 illustrates an exemplary microservice-based design for the backend of an online retailer website [1]. This architecture includes a common backend for frontends pattern, that allows the teams focusing on user-facing side of the application to also handle their own server-side components.

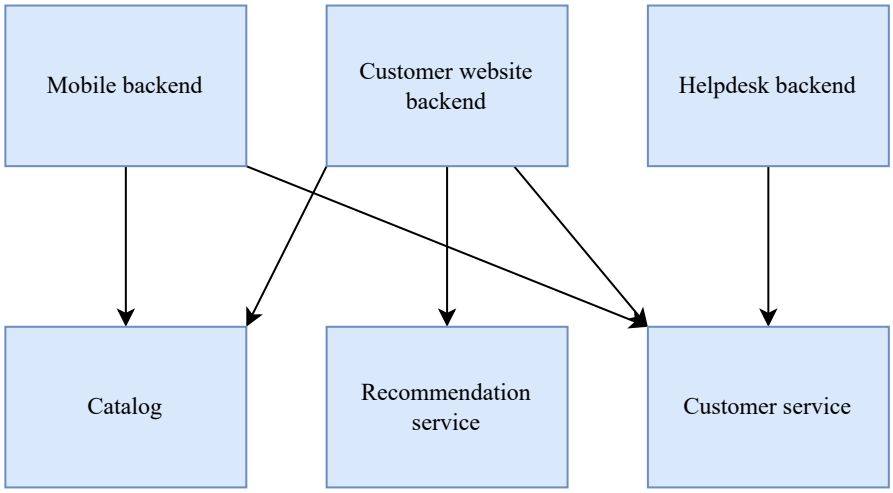

Figure 3. Microservice application example.

The use of containers for microservice architecture has become a widely accepted practice in the industry [1]. This is because containers provide the isolation, portability, and rapid deployment capabilities required to benefit entirely from the microservice

architecture. Reduced resource requirements and startup times of containers, which enable the scalability of microservice applications, are preferred over the improved security of virtual machines.

2.1.4 Kubernetes

Distributed systems running containers are highly volatile. The underlying VMs may fail because of the unpredictable increase in handled data resulting in cascading failures of containers running there. Containers may shut down because of runtime errors or resource shortages in a VM. A container orchestrator is a tool that makes it easier to manage large-scale containerized applications. Container orchestration refers to the automated management of containers, which includes deployment, scaling, and scheduling. Examples of container orchestrators include Kubernetes, Nomad by HashiCorp[14], and Apache Mesos[15].

Container orchestration is necessary because it is challenging to manually manage a large number of containers in a distributed system. Orchestration platforms, such as Kubernetes, provide a high-level API that allows users to manage and scale containers efficiently, abstracting the complexity of container management. These platforms provide features, such as service discovery, load balancing, automatic scaling, self-healing, and fault tolerance, which help ensure the high availability and reliability of containerized applications. Furthermore, container orchestration allows for resource-efficient utilization by optimizing the placement of containers across a cluster of hosts based on resource requirements, availability, and workload demands.

Kubernetes is an open source container orchestration engine for automating the deployment, scaling, and management of containerized applications. Kubernetes operates as a cluster of multiple computing machines referred to as nodes [13]. Each node within a cluster can fulfill the role of a control plane node, a worker node, or perform both functions simultaneously. The control plane makes global decisions about the cluster, as well as detects and responds to cluster events. Worker nodes host the application workload.

Kubernetes runs the application workload inside a set of pods. Pods are the smallest deployable computing units that can be created and managed in Kubernetes. A Pod is a group of one or more containers with shared storage and network resources. Instead of running pods directly, relying on workload resources to create and manage multiple pods is recommended. A controller for the workload resource handles replication, rollout, and automatic healing in case of pod failure. A typical long-running stateless application workload is operated as a Deployment resource [14]. Figure 4 explains an architecture of Kubernetes cluster.

[14] `https://www.nomadproject.io/`
[15] `https://mesos.apache.org/`

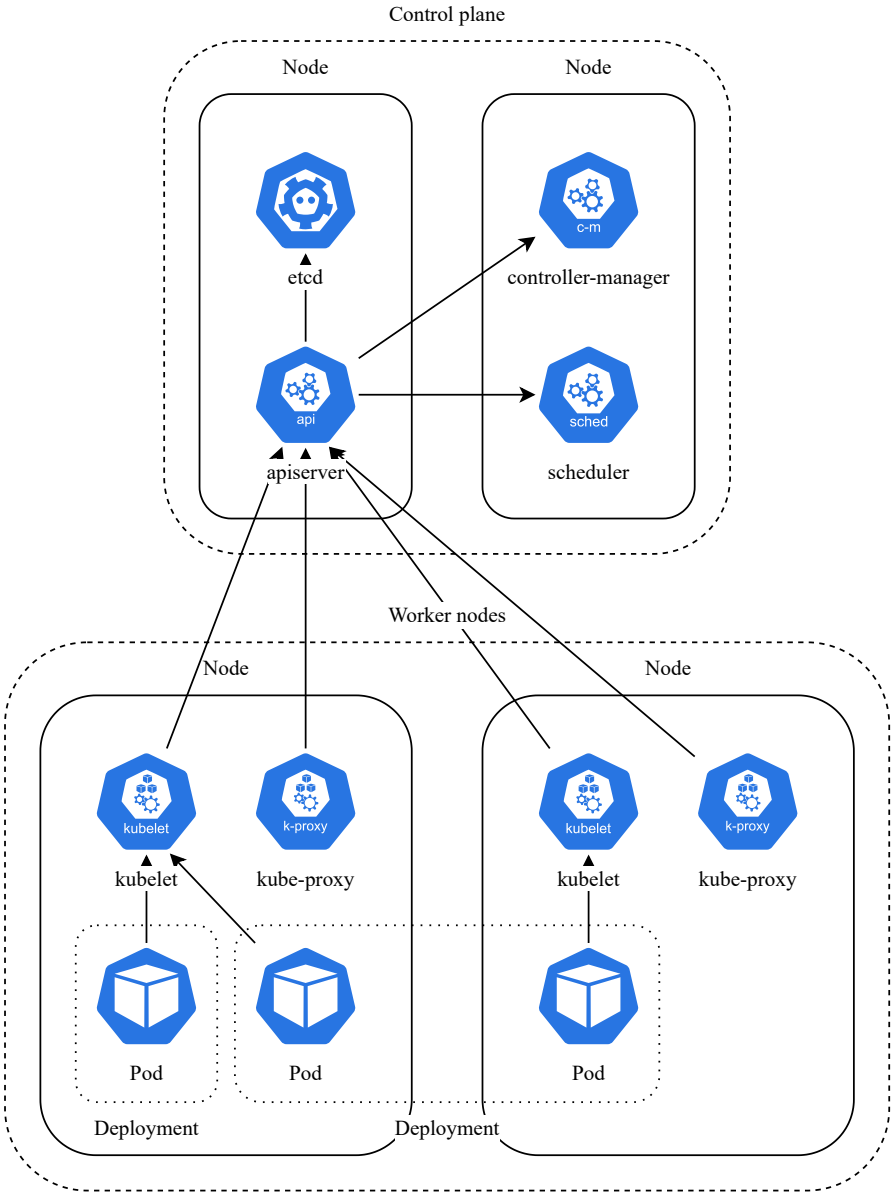

Figure 4. Kubernetes cluster architecture.

2.1.5 Scheduling

Scheduling and placement are two distinct but interconnected concepts within the context of containerized systems. While placement refers explicitly to the mapping of resources to containers, scheduling encompasses a broader scope that includes not only resource allocation, but also the sequencing of container execution. However, within the scope of this thesis, the focus is primarily on long-running containers, in which the order of execution is not a significant concern. As a result, the terms scheduling and placement are used interchangeably throughout this thesis.

In Kubernetes, a scheduler runs as part of the control plane and watches for newly created unassigned pods [15]. First, the scheduler filters out nodes that do not meet the requirements or waits until such node appears. Filtering is based solely on the Pod's requirements. Then the scheduler scores feasible nodes and selects the one with the highest score. If there is more than one node with equal scores, the scheduler randomly selects one of them.

When the number of nodes within a Kubernetes cluster is significant, the scheduler experiences increased time requirements for filtering and scoring nodes. To mitigate the scheduling latency, Kubernetes offers the option of reducing the percentage of nodes to score [16], albeit at the expense of diminished accuracy. Additionally, as the number of nodes in the cluster increases, the likelihood of encountering multiple feasible nodes with equivalent scores amplifies. These nodes, however, may possess inherent inequalities that remain unaccounted for due to the limitations imposed by the scoring rules.

Kubernetes employs an online scheduling model, which involves processing containers individually without considering the other containers awaiting scheduling. Online scheduling focuses on optimizing decisions for each container independently. In contrast, offline scheduling strives to optimize the overall state of the system by considering the collective context [17]. Despite the distinctions between online and offline scheduling algorithms, when confronted with a single container requiring scheduling, both approaches should ideally converge to the same decision.

Figure 5 demonstrates the drawbacks of the online scheduling model through an illustrative scenario. In this scenario, one container has already been scheduled in the cluster of homogeneous virtual machines, while three additional containers of varying sizes await scheduling in the queue. The offline approach utilizes two VMs, whereas the online method, following the order of containers in the queue, employs three VMs, resulting in the underutilization of two of the allocated VMs.

2.2 Related Work

Carmen Carrión [18] explores the role of the scheduler in assigning physical resources to containers, considering various QoS parameters such as response time, energy consumption, and resource utilization optimization. By conducting an empirical study of

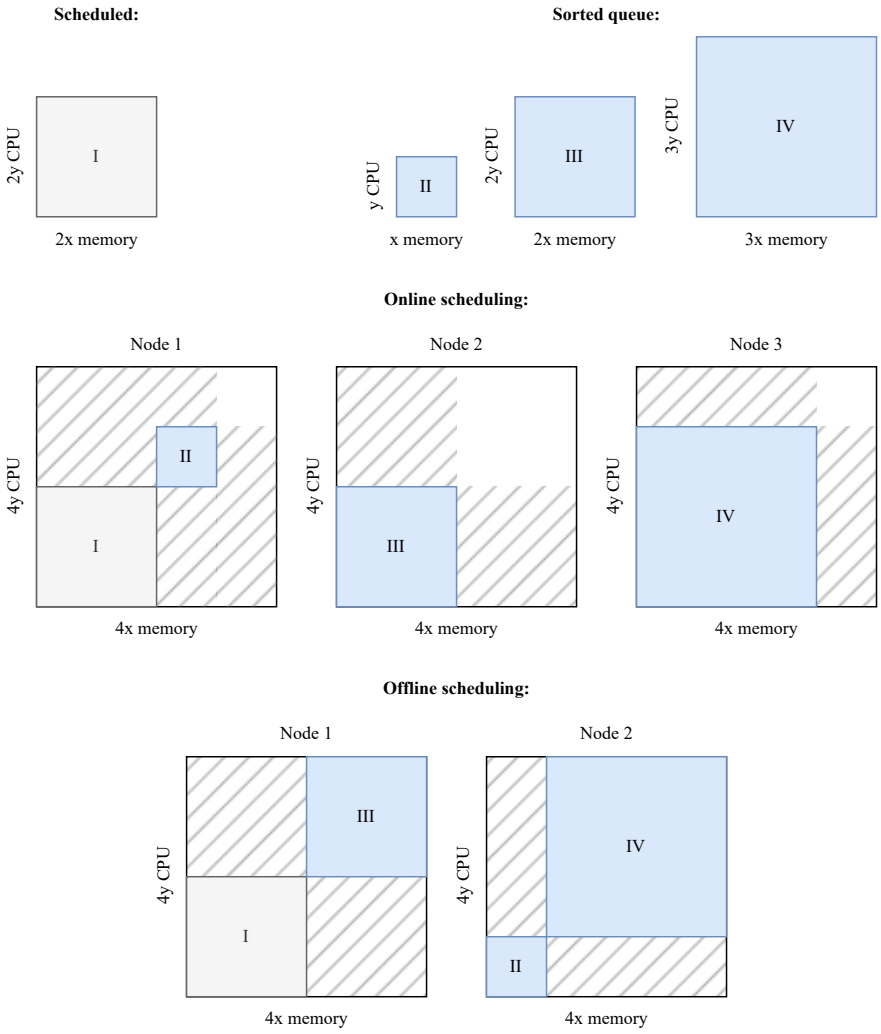

Figure 5. Comparison of online and offline scheduling models.

Kubernetes scheduling techniques, this study offers valuable insights and establishes a new taxonomy for understanding and categorizing these techniques. This study acknowledges the effectiveness of swarm intelligence algorithms for solving scheduling problems, specifically Ant Colony Optimization (ACO), PSO, and Whale Optimization algorithms. Additionally, the gaps identified in current approaches and discussions on challenges, future directions, and research opportunities provide a solid foundation for guiding further research in this area.

Kumar et al. [19] conduct a comprehensive systematic review and classification of scheduling techniques, highlighting their strengths and limitations. The research is

focused on heuristic, meta-heuristic, and hybrid scheduling algorithms. The focus of this thesis, however, is primarily on meta-heuristic algorithms, among which the study explores Particle Swarm Optimization, Ant Colony Optimization, Artificial Honey Bee, and genetic algorithms. The study highlights the advantages of meta-heuristic algorithms, such as their generalized nature, efficiency, and non-deterministic behavior. Conclusively, the research indicates that the PSO algorithm exhibits a superior convergence rate and complexity compared to other meta-heuristic algorithms.

Gang Zhao [20] has devised a modified Particle Swarm Optimization algorithm aimed at optimizing cloud resource utilization, application processing time, and associated costs. The algorithm showed positive outcomes when compared to the conventional PSO algorithm. However, it lacks comprehensive performance evaluation against alternative meta-heuristic or other state-of-the-art algorithms. Finally, the algorithm's consideration does not extend to constrained resource capacities; rather, it exclusively deals with the cost and time functions of running tasks on the resources.

Li et al. [21] leveraged the PSO algorithm in conjunction with a real-time monitoring framework to enhance the Quality of Service. Their study aimed at optimizing node resource utilization, balancing node resources, and minimizing delays between services. The algorithm demonstrates superior QoS optimization compared to the scheduler used by Kubernetes. It is important to note, however, that while the authors implemented an offline model, Kubernetes utilized an online scheduling approach, rendering direct comparisons between the two problematic.

3 Problem Statement

This chapter presents a system model that serves as the basis for subsequent research. The system model encompasses the essential components of the cloud infrastructure and the modelled application. It further outlines the relationships and interactions between these elements to understand the dynamics of microservice placement. Moreover, key assumptions and constraints applied to the system model are explicitly stated to establish the boundaries and scope of the study. To ensure clarity and consistency in subsequent discussions, the relevant notations and formulas that will be used throughout the thesis are introduced in Table 1.

Notation	**Description**
$N = \{N_1, \ldots, N_n\}$	Set of n nodes
N_k	Node at index k
$cost_k$	Cost of node N_k
$cpulim_k$	Effective CPU limit of node N_k
$memlim_k$	Effective memory limit of node N_k
$contlim_k$	Container limit of node N_k
$select(N_k)$	1 if N_k is used by an application, 0 otherwise
$AZ = \{AZ_1, \ldots, AZ_a\}$	Set of a availability zones
AZ_b	Availability zone at index b
$az(N_k)$	Availability zone where node N_k is placed
$place(N_k, AZ_b)$	1 if N_k is placed in AZ_b, 0 otherwise
A	Modeled application
$M = \{M_1, \ldots, M_m\}$	Set of m microservices
M_i	Microservice at index i
$C^i = \{C^i_1, \ldots, C^i_{c_i}\}$	Set of c_i containers of microservice M_i
C^i_j	Container at index j of microservice M_i
$cpureq_i$	CPU requirement of container C^i_j
$memreq_i$	Memory requirement of container C^i_j
$node(C^i_j)$	Node where container C^i_j is scheduled
$sched(C^i_j, N_k)$	1 if C^i_j is scheduled on N_k, 0 otherwise
$data(M_i, M_{i'})$	Network data rate from microservice M_i to $M_{i'}$
$datacost(N_k, N_{k'})$	Cost of data transfer between nodes N_k and $N_{k'}$

Table 1. List of notations.

3.1 Infrastructure Model

N is a set of n nodes $\{N_1, \ldots, N_n\}$ employed by the cluster. Each node N_k is associated with a defined by CSP cost $cost_k$, representing the expenses incurred for running the node over a specific time period, typically one hour. The cost of each node depends on its type, which is uniquely defined by the underlying hardware configuration. The node type specifies the available resources, including CPU, memory, and container capacity.

$$N_k = \langle cost_k;\ cpulim_k;\ memlim_k;\ contlim_k \rangle. \tag{1}$$

The CPU and memory capacity values provided by the CSP are considered nominal, signifying that they are not solely allocated for running user applications. To address potential ambiguity, effective resource limits denoted by $cpulim_k$ and $memlim_k$ are introduced, ensuring that a portion of the nominal resources is reserved for the operating system and container orchestration. Kubernetes provides similar mechanisms for reserving node resources for operational purposes [22]. Reserving the resources also offers the added benefit of mitigating hardware degradation over time [23][24].

The container limit, denoted by $contlim_k$, on a node typically depends on the container orchestrator and cloud service provider. In instances where the container limit value for a node remains unknown, a value of positive infinity ∞ should be used. Consequently, the number of containers scheduled on a node is limited only by the available resource constraints $cpulim_k$ and $memlim_k$. In Kubernetes, however, the limitation is imposed on the number of Pods per node rather than on the number of containers per node. Reference values for the limit of Pods per node for popular Kubernetes distributions are presented in table 2.

Distribution	**Pod limit**
Kubernetes	110 [16]
Amazon Elastic Kubernetes Service (EKS)	737 [17]
Azure Kubernetes Service (AKS)	250 [18]
Google Kubernetes Engine (GKE)	256 [19]

Table 2. Pod limit reference.

[16] https://kubernetes.io/docs/setup/best-practices/cluster-large/
[17] https://github.com/awslabs/amazon-eks-ami/blob/master/files/eni-max-pods.txt
[18] https://learn.microsoft.com/en-us/azure/aks/quotas-skus-regions
[19] https://cloud.google.com/kubernetes-engine/quotas

Each node $N_k \in N$ is assigned to one of the available availability zones, represented as $AZ = \{AZ_1, \ldots, AZ_a\}$. The function $az(N_k)$ returns the specific availability zone where node N_k is placed. Additionally, the boolean function $place(N_k, AZ_b)$ is introduced to determine whether node N_k is placed in the availability zone AZ_b:

$$place(N_k, AZ_b) = \begin{cases} 1 & \text{if } az(N_k) = AZ_b; \\ 0 & \text{if } az(N_k) \neq AZ_b. \end{cases} \tag{2}$$

3.2 Application Model

Application A adheres to the microservice architecture. Figure 3 presents an example of an application following the microservice architecture model. Application A is characterized by a set of long-running stateless m microservices denoted as $M = \{M_1, \ldots, M_m\}$, along with the network relationships between these microservices.

$$A = \langle M;\ data(M_i, M_{i'}) \rangle. \tag{3}$$

Each microservice $M_i \in M$ is defined by its set of containers $C^i = \left\{C^i_1, \ldots, C^i_{c_i}\right\}$, the number of CPU $cpureq_i$ and memory $memreq_i$ resources required by each container:

$$M_i = \left\langle C^i;\ cpureq_i;\ memreq_i \right\rangle. \tag{4}$$

Each container $C^i_j \in C^i$ is assigned to one of the nodes. The function $node\left(C^i_j\right)$ returns the specific node where container C^i_j is scheduled. Additionally, the boolean function $sched\left(C^i_j, N_k\right)$ is introduced to determine whether container C^i_j is scheduled on node N_k:

$$sched\left(C^i_j, N_k\right) = \begin{cases} 1 & \text{if } node\left(C^i_j\right) = N_k; \\ 0 & \text{if } node\left(C^i_j\right) \neq N_k. \end{cases} \tag{5}$$

3.3 Network Model

The data rate from the containers C^i of microservice M_i to the containers $C^{i'}$ of microservice $M_{i'}$ is denoted as $data(M_i, M_{i'})$. In real-world systems, the distribution of requests across multiple servers is typically balanced through mechanisms such as load balancers or Domain Name System (DNS) Round Robin. Thus, it is assumed that network communications are perfectly balanced. Each container $C^i_j \in C^i$ of a microservice M_i both produces and consumes an equal share of network data. Figure 6 provides visual explanation for the network communication between two microservices. Trapezoid in the middle of the figure represents the load balancing entity.

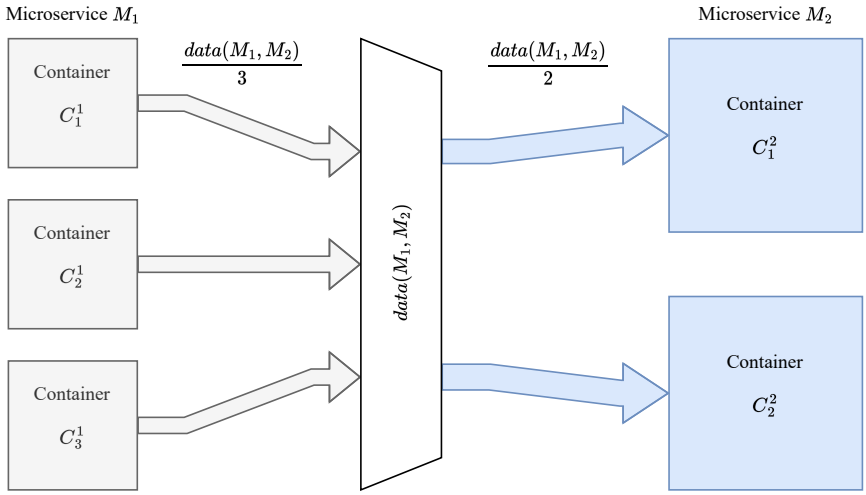

Figure 6. Network communication between two microservices.

In addition the cost associated with running computing machines, cloud service providers typically apply charges for network data transfers between these machines [25]. For this problem, a simplified billing model is adopted. The cost of network data transfer from node N_k to node $N_{k'}$, denoted by $datacost(N_k, N_{k'})$, depends on the logical distance between them: whether it is the same node, the nodes are in the same, or different availability zones. Notably, data transfer between containers running on the same host node incurs no additional charges. While network data transfer is conventionally measured in bits per second, to ensure clarity and consistency, it is advisable to reuse the time unit of $cost_k$ for both $data(M_i, M_{i'})$ and $datacost(N_k, N_{k'})$.

$$datacost(N_k, N_{k'}) = \begin{cases} 0 & \text{if } k = k'; \\ \alpha \geq 0 & \text{if } az(N_k) = az(N_{k'}); \\ \beta \geq \alpha & \text{if } az(N_k) \neq az(N_{k'}). \end{cases} \tag{6}$$

As the application users are not situated within the CSP's DCs, their traffic is considered external to the system. Although this external traffic incurs costs, it cannot be affected by any changes in container placement. As a result, this external traffic is not factored into the system model.

The network traffic is constrained by the network bandwidth available within the system. In real-world scenarios, the network bandwidth is typically defined in Gigabits per second on a VM basis, considering both inbound and outbound communications separately [26]. In most cases, the provided network bandwidth values are sufficient. In other cases, CSPs offer options to achieve higher bandwidth, for example by switching to

larger VM types. Given the high base numbers and the possibility to upscale bandwidth as needed, network bandwidth is not considered within the system model.

3.4 Kubernetes Model

While Kubernetes does not directly operate in terms of data transfer $data(M_i, M_{i'})$ or infrastructure costs $cost(N_k)$ and $datacost(N_k, N_{k'})$, it is possible to align the system model of this thesis with Kubernetes API. Table 3 provides a mapping of notations discussed in the chapter to the Kubernetes notations. The utilization of the provided table streamlines the application of the solution presented in the thesis, enabling the development of a custom scheduler or scheduling plugin for Kubernetes.

System model	**Kubernetes**
Node	Node
Nominal resource capacity	Resource capacity
Reserved resource capacity	Reserved resource
Effective resource limit	Allocatable resource
Availability zone	Zone topology domain
Microservice	Deployment
Container	Pod
Resource requirement	Resource requirement

Table 3. System model to Kubernetes notation mapping.

3.5 Problem Objective

The optimization of container placement aims to minimize the overall cost of running the application in the cloud. The objective model omitted several common objectives:

- Energy efficiency is crucial for cloud providers to cut utility costs but is irrelevant to customers because of the billing model.
- Balancing resource utilization, which boosts QoS and hardware lifespans for providers, is achieved via resource reservation.
- Performance, as each solution satisfying the constraints ensures the scheduled containers have sufficient resources for operation.

3.5.1 Objective function

$$\text{minimize}(COST = NODECOST + DATACOST). \tag{7}$$

The cost of running the application $COST$ consists of two components: the cost of operating computing machines $NODECOST$ and the cost of network data transfer between these machines $DATACOST$.

$$NODECOST = \sum_{k=1}^{n}(cost_k \times select(N_k)). \tag{8}$$

Despite the presence of n nodes in the cluster, it remains possible to rebalance containers between them in a manner that renders a particular node N_k empty, subsequently allowing its removal from the cluster. The function $select(N_k)$ is introduced to determine whether node N_k is utilized by application A.

$$select(N_k) = \text{sgn}\left(\sum_{i=1}^{m}\sum_{j=1}^{c_i} sched\big(C_j^i, N_k\big)\right). \tag{9}$$

$$\forall M_i \forall N_k, \quad sched\big(C_j^i, N_k\big) \geq 0 \Rightarrow select(N_k) \geq 0, \quad \forall N_k. \tag{10}$$

The total cost of network data transfers between the microservices $DATACOST$ is equivalent to the sum of the costs incurred for each node-to-node data transfer $DATA(N_k, N_{k'})$.

$$DATACOST = \sum_{k=1}^{n}\sum_{k'=1}^{n}(datacost(N_k, N_{k'}) \times DATA(N_k, N_{k'})). \tag{11}$$

To calculate the amount of data transferred from node N_k to node $N_{k'}$, it is necessary to aggregate all the network data transfer shares produced by the containers present on the node N_k and consumed by containers present on the node $N_{k'}$.

$$DATA(N_k, N_{k'}) = \sum_{i=1}^{m}\sum_{i'=1}^{m}\left(data(M_i, M_{i'}) \times \sum_{j=1}^{c_i}\frac{sched\big(C_j^i, N_k\big)}{c_i} \times \sum_{j'=1}^{c_{i'}}\frac{sched\big(C_{j'}^{i'}, N_{k'}\big)}{c_{i'}}\right). \tag{12}$$

3.5.2 Constraints

$$\sum_{k=1}^{n} sched\big(C_j^i, N_k\big) = 1, \qquad \forall M_i, \forall C_j^i \qquad (13)$$

$$\sum_{i=1}^{m} \sum_{j=1}^{c_i} sched\big(C_j^i, N_k\big) \leq contlim_k, \qquad \forall N_k, \qquad (14)$$

$$\sum_{i=1}^{m} \sum_{j=1}^{c_i} \big(sched\big(C_j^i, N_k\big) \times cpureq_i\big) \leq cpulim_k, \qquad \forall N_k, \qquad (15)$$

$$\sum_{i=1}^{m} \sum_{j=1}^{c_i} \big(sched\big(C_j^i, N_k\big) \times memreq_i\big) \leq memlim_k, \qquad \forall N_k. \qquad (16)$$

- Constraint (13) reflects the requirement that each container C_j^i of any microservice M_i is assigned to exactly one node.
- Constraint (14) ensures that for any node N_k, the number of containers assigned to it is lower or equal to $contlim_k$.
- Constraint (15) ensures that for any node N_k, the amount of CPU resources utilized by the containers assigned to it is lower or equal to its effective CPU limit $cpulim_k$.
- Respectively, constraint (16) ensures that for any node N_k, the amount of memory resources utilized by the containers assigned to it is lower or equal to its effective memory limit $memlim_k$.

3.6 Complexity

The well-established nondeterministic polynomial-hard vector bin packing problem [27] can be trivially reduced to a container scheduling problem to prove the nondeterministic polynomial-hardness of the latter. Each multidimensional vector in the vector bin packing problem has the same characteristics as the containers within the container scheduling problem, whereas the bins are aligned with the nodes.

4 Solution

This chapter introduces the solution, comprising a detailed explanation of the proposed algorithm and its implementation. Additionally, a comprehensive exploration of the design choices made during the development process and the underlying reasons that influence these decisions are provided.

4.1 Particle Swarm Optimization

James Kennedy and Russell Eberhart [28] initially introduced the concept of Particle Swarm Optimization as a technique for optimizing continuous nonlinear functions. This algorithm replicates the social behaviors observed in swarms, where individual particles move towards both their own best known positions and the best known position of the entire swarm. The elegance of the algorithm lies in its simplicity, relying on primitive mathematical operations, while also demonstrating efficiency in terms of memory usage and processing speed. The optimizer performed well on genetic algorithm test functions and training artificial neural network weights.

Recent surveys investigating scheduling techniques in Kubernetes [18] and cloud environments [29][30][31][32][33][19] highlighted the viability of Particle Swarm Optimization or PSO-based solutions. However, it is important to recognize that scheduling lacks a singular definitive solution owing to its NP-hard nature, and each approach has distinct advantages and disadvantages. Notably, a significant limitation of PSO is its attractiveness to local optima, because the global optimum is not adjusted in every iteration. Consequently, the convergence rate of PSO is relatively slow, although it often outperforms other meta-heuristic algorithms. The strengths of PSO include the following:

- simplicity of implementation and extension;
- absence of complex mathematical operations such as derivatives.
- dependency on a few easily adjustable parameters.

The ease of extensibility of the algorithm plays a crucial role in its popularity because incorporating greedy or other meta-heuristic algorithms into PSO minimizes its drawbacks.

PSO operates on particles $P = \{P_1, \ldots, P_n\}$ composing a swarm. Each particle P_i is characterized by the positions X_i^d and velocities V_i^d in each $1 \leq d \leq D$ dimension of D-dimensional search space. The objective function is applied to the position of each particle to calculate the worth of the solution. The goal of the algorithm is to determine the position of the particle with the best value of the objective function. During the initialization procedure, every particle is assigned a random position and random velocity to travel in the search space. In each iteration every particle updates its velocity and

position $X_i^d = X_i^d + V_i^d$. Depending on the values of the objective function, personal (local) best known position $Pbest_i$ and swarm's (global) best known position $Gbest$ searched by the whole swarm is updated in every iteration. Original Particle Swarm Optimization algorithm implementation is presented in Algorithm 1.

Algorithm 1: Original Particle Swarm Optimization algorithm [28].

Input: objective function `f()`, number of particles n, number of dimensions D, termination criterion T

Result: best known position $Gbest$ in the swarm

foreach *particle* $i = 1, \ldots, n$ **do**
  foreach *dimension* $d = 1, \ldots, D$ **do**
    $V_i^d \leftarrow$ `random()`
    $X_i^d \leftarrow$ `random()`
  $Pbest_i \leftarrow X_i$
  if `f(`$Pbest_i$`) > f(`$Gbest$`)` **then**
    $Pbest_i \leftarrow Gbest$

while *not* T **do**
  foreach *particle* $i = 1, \ldots, n$ **do**
    foreach *dimension* $d = 1, \ldots, D$ **do**
      $V_i^d \leftarrow V_i^d + 2 \times$ `random()` $\times (Pbest_i^d - X_i^d) + 2 \times$ `random()` $\times (Gbest^d - X_i^d)$
      $X_i^d \leftarrow X_i^d + V_i^d$
    if `f(`X_i`) > f(`$Pbest_i$`)` **then**
      $Pbest_i \leftarrow X_i$
      if `f(`$Pbest_i$`) > f(`$Gbest$`)` **then**
        $Gbest \leftarrow Pbest_i$

Yuhui Shi and Russell Ebenhart [34] modified the original PSO algorithm introducing the inertia weight parameter w, updating the formula for particle's velocity calculation:

$$V_i^d = w \times V_i^d + c_1 \times r_1 \times (Pbest_i^d - X_i^d) + c_2 \times r_2 \times (Gbest^d - X_i^d), \qquad (17)$$

where $0 \leq c_1, c_2 \leq 2.5$ are acceleration coefficients and $r_1, r_2 \in [0, 1]$ are random values. Simulations showed that optimizer with inertia weight in the range $[0.9, 1.2]$ on average performs better. Furthermore, a time decreasing inertia weight brings in a significant improvement on the PSO performance.

4.2 Proposed Solution

To tailor the algorithm to the problem, each container is represented by a dimension in the search space, while every node corresponds to a point across each dimension. Consequently, the particle positions within the search space are the mappings of nodes to containers, essentially representing solutions to the problem. This representation adheres to the constraint (13) that each container is exclusively assigned to a single node, yet a single node may accommodate multiple containers. From the proposed model, it is evident that the total number of solutions, including both feasible and infeasible solutions due to constraints, is equal to $\prod_{i=1}^{m} n^{c_i} = n^{\sum_{i=1}^{m} c_i}$. An illustrative depiction of the problem input data is presented in Table 4, and the corresponding search space is shown in Figure 7.

Variable	Value
n	4
$cost_k$	$[1, 1.9, 3.61, 6.86]$
$cpulim_k$	$[1, 2, 4, 8]$
$memlim_k$	$[512, 1024, 2048, 4096]$
$contlim_k$	$[2, 4, 8, 16]$
m	2
c_i	$[1, 1]$
$cpureq_i$	$[2, 1]$
$memreq_i$	$[512, 1024]$

Table 4. Problem input example.

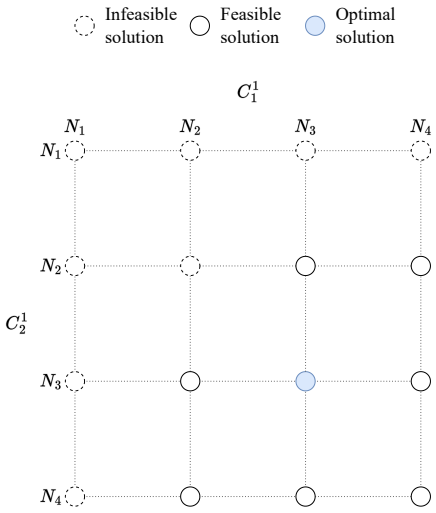

Figure 7. Search space in Particle Swarm Optimization algorithm.

Employing PSO within the context of the established system model presents a set of challenges. The initial challenge arises from the nature of the search space. PSO was initially formulated to function within continuous search spaces, whereas the current space is inherently discrete. Adding to this, the constraints (14)-(16) imply that not every point within the space corresponds to a feasible solution. This challenge is particularly evident during the particle initialization stage. The random generation of particle positions frequently fails to yield feasible solutions, resulting in the majority of particles lacking a feasible best known position $Pbest_i$. Additionally, when the number of particles is small, the probability of no particles finding a feasible position during the random initialization process is high, leading to the absence of the swarm's best known position $Gbest$. This deficiency significantly affects the velocity update process

during each iteration. A heuristic bin packing algorithm First-Fit is proposed to initialize feasible positions within a search space.

Algorithm 2 outlines the feasible position initialization procedure. For each container, the first-fit algorithm searches through the nodes to identify the first node with sufficient available capacity. The first-fit algorithm (lines 3-7) executed for every particle to generate a feasible solution without any modifications will lead to identical initial positions across particles. To mitigate this, nodes are shuffled for each particle, ensuring distinct node orders for each first-fit instance.

Algorithm 2: Feasible position initialization algorithm.

Input: particles P, containers C, nodes N
Result: position vectors X

```
1 foreach particle p in P do
2     shuffle N
3     foreach container c in C do
4         foreach node n in N do
5             if n has sufficient available capacity for c then
6                 decrease the available capacity of the n
7                 set X_p[c] to the index of n
```

A subsequent challenge pertains to the dimensionality of the search space. Real-world scenarios typically involve a significantly larger number of containers than nodes, resulting in a highly-dimensional yet constrained search space. This tightness often causes particles to traverse beyond the boundaries of the space. Addressing this issue necessitates the implementation of restricted boundary conditions to ensure that particle movement remains within acceptable limits.

4.3 Complexity Analysis

The first-fit algorithm sequentially scans each node to identify the first one capable of accommodating a container. This process is reiterated for every container, resulting in a time complexity of $O(nc)$, with n denoting the number of nodes and c - the number of containers.

The proposed PSO-based algorithm initializes each particle's position using the first-fit algorithm, yielding a time complexity of $O(p) \times O(nc) = O(pnc)$, where p is the particle count. Subsequently, the algorithm updates the velocities, positions, and best known positions of each particle. The velocity and position updates for a single dimension require constant time $O(1)$, resulting in an update process duration of

$O(c)$. For the objective function, calculating the running machine costs takes $O(n)$ time, while computing data transfer costs involves iterating over each potential node pair and producer-consumer microservice pair, resulting in a time complexity of $O(n^2m^2)$, where m is the microservice count. Consequently, the total time complexity of the objective function is $O(n) + O(n^2m^2) = O(n^2m^2)$. Summing these complexities, the overall time complexity of the algorithm is $O(pnc) + O(i) \times O(p) \times (O(c) + O(n^2m^2)) = O(pnc) + O(ip \times (c + n^2m^2)))$, where i denotes the number of iterations.

Throughout its execution, the algorithm retains the complete problem input in memory—nodes, microservices, and the data rate lookup table. The first-fit algorithm does not require an auxiliary space. During the PSO phase of the proposed algorithm, each particle tracks its velocity vector, position vector, and the best known position. Consequently, the algorithm's overall space complexity is $O(n)+O(m)+O(m^2)+O(pc)+O(pc)+O(p) = O(n + m^2 + pc)$, where the auxiliary space occupies $O(pc)$.

5 Experiments

This chapter describes the experiments conducted, including the experimental setup, metrics that were quantified, methodologies employed for measurement, and results.

5.1 Setup

The experiments were conducted using an ASUS ROG Flow X13 GV301QE-K6065 laptop connected to a power source. The laptop is equipped with an AMD Ryzen 9 5900HS (3.0 - 4.6 GHz) CPU and 32 GiB of RAM.

The proposed algorithm is evaluated against a CP-SAT [20] on inputs which take CP-SAT a reasonable time to finish. CP-SAT is an award-winning linear solver from Google OR-Tools software suite [35]. The CP-SAT solver is implemented as a lazy clause generation solver on top of a SAT solver.

Experiments were conducted on randomly generated scenarios that closely mimicked real-world applications. Microservices exhibit multiples of 250 milli central processing units for $cpureq_i$ and 128 MiB random access memory for $memreq_i$. The nodes encompass integer values of central processing units for $cpulim_k$ and multiples of 512 MiB random access memory for $memlim_k$. $cost_k$ and $contlim_k$ depend on the generated values of $cpulim_k$ and $memlim_k$. The network data communication graph follows a tree-like structure.

5.2 Results

Experiments were conducted to assess the performance and accuracy of the proposed algorithms. The execution time, recorded in seconds, exclusively accounts for the solving phase, excluding the input reading and output provision durations. Evaluating accuracy poses a challenge because of the inherent randomness and variability of Particle Swarm Optimization as a meta-heuristic algorithm. To mitigate this, the seed value used by the random number generator was fixed to one of three values, repeated for each experiment, enabling the selection of the optimal outcome.

The comparison of constraint programming-boolean satisfiability and the suggested Particle Swarm Optimization-based solution results is illustrated in Table 5. The execution time of the proposed algorithm is heavily dependent on the number of iterations, yet it is considerably lower than that of the linear solver. Elevating the iteration count does not increase accuracy, as the particles tend to become entrapped in local optima. This situation can be observed by monitoring the iteration at which the last update to the swarm's best position occurred. Figure 8 demonstrates the percentage of containers scheduled by the proposed algorithm suboptimally in comparison to CP-SAT.

[20] `https://developers.google.com/optimization/cp/cp_solver`

	CP-SAT		PSO	
Input	**Time**	**Value**	**Time**	**Value**
$m = 4\ c_i = 7\ n = 15$	20 seconds	304.17	0.1 second	442.62
$m = 4\ c_i = 8\ n = 14$	51 seconds	458.40	0.1 second	538.59
$m = 5\ c_i = 4\ n = 40$	83 seconds	305.26	0.1 second	437.17
$m = 6\ c_i = 6\ n = 15$	548 seconds	836.23	0.1 second	1007.37
$m = 9\ c_i = 5\ n = 15$	189 seconds	798.92	0.1 second	952.06

Table 5. Comparison of CP-SAT and proposed PSO-based solutions.

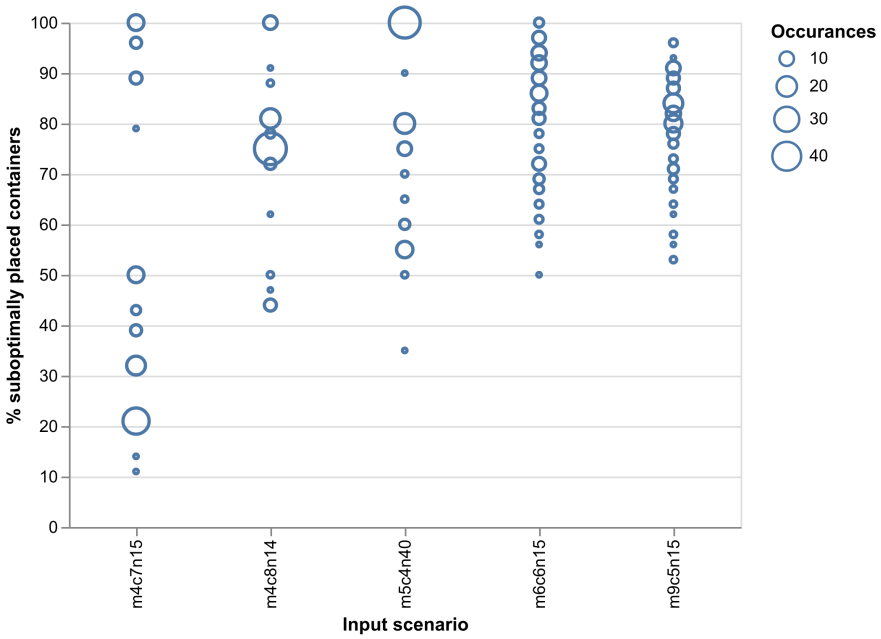

Figure 8. Percentage of suboptimally placed containers.

Table 6 shows the impact of applying the first-fit algorithm to initialize the particle position on the discovery of at least one feasible solution. The experiment used identical input scenarios and parameters for both algorithms, with the sole distinction being the use of first-fit or random values for the position initialization. Each algorithm was run 100 times per input scenario and the number of feasible solutions found was calculated. This approach takes advantage of the inherent randomness of the meta-heuristic PSO algorithm. The use of the first-fit algorithm always ensures the discovery of at least one feasible solution. Another advantage of using first-fit, which was not explored in this particular experiment, lies in making the proposed algorithm deterministic for small-scale scenarios.

Input	PSO	PSO+FF
$m = 4\ c_i = 7\ n = 15$	100	100
$m = 4\ c_i = 8\ n = 14$	39	100
$m = 5\ c_i = 4\ n = 40$	90	100
$m = 6\ c_i = 6\ n = 15$	100	100
$m = 9\ c_i = 5\ n = 15$	0	100
random $m \in [20, 40]\ c_i \in [1, 10]\ n \in [50, 150]$	4	100

Table 6. Effect of using first-fit on finding at least one feasible solution.

The multiple simulated runs revealed several distinct patterns. The proposed algorithm exhibits rapid convergence, albeit frequently becoming trapped in local optima. In particular, when the difference between the number of containers and nodes is not substantial, the first-fit algorithm used for particle position initialization, which is repeated for each particle, tends to uncover a local optimum that serves as the point of particle convergence. Therefore, the optimum was found within the initial iterations and remained independent of the number of particles.

6 Conclusion

This thesis addresses the complex problem of optimizing the placement of microservices in a cloud environment. By proposing a novel particle swarm optimization PSO-based algorithm and integrating network data transfer costs into the objective function, this research explored an innovative approach to minimizing cloud infrastructure costs.

The experimental results, which compared the proposed algorithm with a linear solver and the standard Particle Swarm Optimization algorithm, demonstrated its efficiency and accuracy while also revealing the limitations common to meta-heuristic algorithms. The proposed modifications were proven to be effective in increasing the likelihood of identifying feasible solutions when compared with the standard PSO. The algorithm's capability to rapidly converge highlights its effectiveness even though it occasionally gets trapped in local optima.

A limitation of this thesis is the assumption of application stability and disregarding scalability. For instance, if the user-generated load suddenly increases by $X\%$, the existing nodes might require augmentation, potentially rendering the prior node-container assignments suboptimal. Similarly, an $X\%$ load decrease may not justify node removal. These scenarios encourage searching for optimal solutions that satisfy both the initial and scaled states. Further potential research directions involve system models that consider such fluctuations, or accommodate stateful or short-lived containers.

References

[1] Mike Loukides, Steve Swoyer. O'Reilly Media, "Microservices Adoption in 2020." https://www.oreilly.com/radar/microservices-adoption-in-2020/. Accessed: 2023-04-16.

[2] B. Franklin, "40 cloud computing stats and trends to know in 2023." https://cloud.google.com/blog/transform/top-cloud-computing-trends-facts-statistics-2023. Accessed: 2023-04-16.

[3] C. N. C. Foundation, "CNCF Annual Survey 2022." https://www.cncf.io/reports/cncf-annual-survey-2022/. Accessed: 2023-04-10.

[4] Datadog, "9 insights on real world container use." https://www.datadoghq.com/container-report/. Accessed: 2023-04-10.

[5] T. Balakrishnan, C. Gnanasambandam, L. Santos, and B. Srivathsan, "Cloud-migration opportunity: Business value grows, but missteps abound." https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/cloud-migration-opportunity-business-value-grows-but-missteps-abound. Accessed: 2023-08-04.

[6] Vantage, "Cloud Cost Report: Q2 2023." https://www.vantage.sh/cloud-cost-report/2023-q2. Accessed: 2023-08-04.

[7] P. Mell and T. Grance, "The nist definition of cloud computing," Tech. Rep. NIST SP 800-145, National Institute of Standards and Technology, Gaithersburg, MD, Sept. 2011.

[8] Amazon Web Services, Inc., "Global infrastructure." https://aws.amazon.com/about-aws/global-infrastructure/. Accessed: 2023-04-03.

[9] K. Scarfone, M. Souppaya, and P. Hoffman, "Guide to security for full virtualization technologies," Tech. Rep. NIST SP 800-125, National Institute of Standards and Technology, Gaithersburg, MD, 2008.

[10] Amazon Web Services, Inc., "What is a Hypervisor?." https://aws.amazon.com/what-is/hypervisor/. Accessed: 2023-05-04.

[11] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide," Tech. Rep. NIST SP 800-190, National Institute of Standards and Technology, Gaithersburg, MD, Sept. 2017.

[12] S. Newman, *Building microservices: designing fine-grained systems*. Beijing Sebastopol, CA: O'Reilly Media, first edition ed., 2015. OCLC: ocn881657228.

[13] Kubernetes, "Kubernetes Components." https://kubernetes.io/docs/concepts/overview/components/. Accessed: 2023-07-03.

[14] Kubernetes, "Kubernetes Pods." https://kubernetes.io/docs/concepts/workloads/pods/. Accessed: 2023-07-03.

[15] Kubernetes, "Kubernetes Scheduler." https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. Accessed: 2023-04-10.

[16] Kubernetes, "Scheduler Performance Tuning." https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning/. Accessed: 2023-08-04.

[17] L. F. Bittencourt, A. Goldman, E. R. Madeira, N. L. da Fonseca, and R. Sakellariou, "Scheduling in distributed systems: A cloud computing perspective," *Computer Science Review*, vol. 30, pp. 31–54, 2018.

[18] C. Carrión, "Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges," *ACM Computing Surveys*, vol. 55, no. 7, pp. 138:1–138:37, 2022.

[19] M. Kumar, S. Sharma, A. Goel, and S. Singh, "A comprehensive survey for scheduling techniques in cloud computing," *Journal of Network and Computer Applications*, vol. 143, pp. 1–33, 2019.

[20] G. Zhao, "Cost-aware scheduling algorithm based on pso in cloud computing environment," *International Journal of Grid and Distributed Computing*, vol. 7, no. 1, pp. 33–42, 2014.

[21] H. Li, X. Wang, S. Gao, and N. Tong, "A Service Performance Aware Scheduling Approach in Containerized Cloud," in *2020 IEEE 3rd International Conference on Computer and Communication Engineering Technology (CCET)*, (Beijing, China), pp. 194–198, IEEE, Aug. 2020.

[22] Kubernetes, "Reserve Compute Resources for System Daemons." https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/. Accessed: 2023-07-10.

[23] S. Malkowski, Y. Kanemasa, H. Chen, M. Yamamoto, Q. Wang, D. Jayasinghe, C. Pu, and M. Kawaba, "Challenges and opportunities in consolidation at high resource utilization: Non-monotonic response time variations in n-tier applications," in *2012 IEEE Fifth International Conference on Cloud Computing*, pp. 162–169, 2012.

[24] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2012.

[25] B. Pal, S. Gorczynski, and D. Schmidt, "Overview of Data Transfer Costs for Common Architectures | AWS Architecture Blog." https://aws.amazon.com/blogs/architecture/overview-of-data-transfer-costs-for-common-architectures/. Accessed: 2023-08-07.

[26] Amazon Web Services, Inc., "Amazon EC2 instance network bandwidth - Amazon Elastic Compute Cloud." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html. Accessed: 2023-08-06.

[27] K. Maruyama, S. K. Chang, and D. T. Tang, "A general packing algorithm for multidimensional resource requirements," *International Journal of Computer & Information Sciences*, vol. 6, pp. 131–149, June 1977.

[28] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948 vol.4, 1995.

[29] G. Singh and A. K. Chaturvedi, "Particle swarm optimization-based approaches for cloud-based task and workflow scheduling: A systematic literature review," in *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*, pp. 350–358, 2021.

[30] I. Ahmad, M. G. AlFailakawi, A. AlMutawa, and L. Alsalman, "Container scheduling techniques: A survey and assessment," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 7, pp. 3934–3947, 2022.

[31] M. T. Mon and M. A. Khine, "Scheduling and load balancing in cloud-fog computing using swarm optimization techniques: A survey," 2019.

[32] M. Adhikari, T. Amgoth, and S. N. Srirama, "A survey on scheduling strategies for workflows in cloud environment and emerging trends," *ACM Comput. Surv.*, vol. 52, aug 2019.

[33] A. Arunarani, D. Manjula, and V. Sugumaran, "Task scheduling techniques in cloud computing: A literature survey," *Future Generation Computer Systems*, vol. 91, pp. 407–415, 2019.

[34] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, pp. 69–73, 1998.

[35] G. for Developers, "OR-Tools." `https://developers.google.com/optimization`. Accessed: 2023-08-11.

Appendix

I. Source Code

The source code for this thesis is accessible from a public GitHub [21] repository: `https://github.com/chernetskyi/container-placement/`.

[21] `https://github.com/`

II. Methodology

To enhance the overall writing quality and academic style of this thesis, AI-powered writing assistants, Grammarly [22] and Paperpal [23], were incorporated into the writing process. Grammarly, accessed through the Grammarly for Education plan via its website, was used to address grammar and punctuation issues within the selected paragraphs. Additionally, Paperpal, accessed via the free plan on its website, was used to improve the academic style of specific text segments. These writing assistants served as valuable tools for refining the language, coherence, and clarity of the thesis.

[22] https://www.grammarly.com/
[23] https://paperpal.com/

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Volodymyr Chernetskyi**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Container-Based Microservice Placement Optimization in Cloud,
   *(*title of thesis)

   supervised by Chinmaya Kumar Dehury.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Volodymyr Chernetskyi
11/08/2023