universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Simulating Dynamic Cloud Marketspaces: Modeling Spot Instance Behavior and Scheduling with CloudSim Plus

verfasst von | submitted by

Christoph Goldgruber BSc (WU)

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 926
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von | Supervisor:	ao. Univ.-Prof. i.R. tit. Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta
Mitbetreut von | Co-Supervisor:	Dipl.-Ing. Dr. Benedikt Pittl BSc

Abstract

The increasing reliance on dynamic pricing models, such as spot instances, in public cloud environments presents new challenges for workload scheduling and reliability. While these models offer cost advantages, they introduce volatility and uncertainty that are not fully addressed by current allocation algorithms or simulation tools.

This thesis contributes to the modeling and evaluation of such environments by extending the CloudSim Plus simulation framework to support realistic spot instance lifecycle management, including interruption, termination, hibernation, and reallocation. The enhanced simulator is validated using synthetic scenarios and large-scale simulations based on the Google Cluster Trace dataset.

Building on this foundation, the HLEM-VMP allocation algorithm—originally proposed in earlier research—was adapted to operate under dynamic spot market conditions. Its performance was evaluated against baseline allocation strategies to assess its efficiency and resilience in volatile workload environments. The comparison demonstrated a reduction in the number of spot instance interruptions as well as a decrease in the maximum interruption duration.

Overall, this work provides both a simulation framework for simulating dynamic cloud behavior and analytical insights into virtual machine allocation performance and market risk, contributing to more robust and cost-effective resource management in cloud computing.

Kurzfassung

Die zunehmende Abhängigkeit von dynamischen Preismodellen wie Spot-Instanzen in öffentlichen Cloud-Umgebungen stellt neue Herausforderungen für das Scheduling und die Zuverlässigkeit von Cloud-Workloads dar.

Diese Masterarbeit leistet einen Beitrag zur Modellierung und Bewertung solcher Umgebungen, indem das CloudSim-Plus-Simulationsframework erweitert wird, um einen realistischen Lebenszyklus von Spot-Instanzen zu unterstützen – einschließlich Unterbrechung, vorzeitiger Beendigung, zwischenzeitlichem Ruhezustand (Hibernation) und anschließender Re-Allokation von Instanzen. Die Erweiterung des Frameworks wird sowohl durch synthetische Szenarien als auch durch Simulationen auf Basis des Google-Cluster-Trace-Datensatzes validiert.

Aufbauend auf dieser Grundlage wird ein aus der Fachliteratur entnommener Allokationsalgorithmus, HLEM-VMP, an die Bedingungen dynamischer Spot-Märkte angepasst. Seine Leistungsfähigkeit wird mit Basisstrategien verglichen, um Effizienz und Robustheit bei volatilen Cloud-Workloads zu evaluieren. Der Vergleich zeigte eine Reduktion der Anzahl an Unterbrechungen von Spot-Instanzen sowie eine Verringerung der maximalen Unterbrechungsdauer.

Insgesamt liefert diese Arbeit sowohl ein Simulationsframework zur Simulation dynamischer Cloud-Umgebungen als auch analytische Erkenntnisse zur Effektivität von Allokationsstrategien und zum Umgang mit Marktrisiken. Sie trägt damit zu einer robusteren und kosteneffizienteren Ressourcenverwaltung im Cloud Computing bei.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1 Introduction

The idea of making computing resources a public utility was first introduced in 1961 by John McCarthy. In the late 1990s, the first service providers emerged that attempted to implement this concept [6, p. 2]. However, cloud computing only began to gain popularity in 2007 [7, p. 173]. This rise in popularity can be attributed to an initiative by Google and IBM, which introduced cloud computing programs to many universities in the US and, subsequently, to institutions worldwide [8, 6].

Today, cloud computing is an integral part of several technologies such as mobile Internet, automation of knowledge work, big data, and the Internet of Things [9, p. 5]. In 2024, cloud infrastructure providers collectively generated a revenue of around $330 billion [10]. The total economic impact of cloud computing technologies is estimated to be significantly higher. Some sources estimate that their contribution to the global economy will exceed $1.7 trillion annually by 2025 [9, p. 5]. A widely accepted definition of cloud computing was provided by the National Institute of Standards and Technology (NIST), which states the following [11]:

> Cloud computing is a model for enabling ubiquitous, convenient, 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.

Cloud computing is characterized by five essential features: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service [11, p. 2].

On-demand self-service means that users can access computing capabilities whenever needed, without requiring human interaction with the cloud provider. Broad network access implies that users can utilize cloud services over the Internet, and that these services can be accessed from various device types. Resource pooling indicates that physical and virtual resources in a cloud environment are shared among multiple users and dynamically allocated according to demand. Users typically have no control over or knowledge of the exact location of the provided resources, although they may be able to specify parameters such as the country or data center. Rapid elasticity refers to the ability to automatically scale resources up or down based on user demand. Lastly, measured service means that resource usage is monitored, controlled, and reported, allowing for optimized usage for both providers and consumers [11, p. 2]. The common cloud computing service models are generally divided into three categories [9, 11]:

Software as a Service (SaaS). In this service model, users are provided with applications delivered over the Internet [6, p. 4]. The software runs on cloud infrastructure and is

accessed via a client interface, such as a web browser [7, p. 174]. Examples of SaaS offerings include Microsoft Office 365, Dropbox, and Google Docs.

Platform as a Service (PaaS). In the PaaS model, users receive all necessary components for developing new applications [6, p. 4]. This includes environments for development, testing, deployment, and hosting. By providing a wide array of readily available tools, PaaS can significantly reduce development time [9, p. 6]. Examples of PaaS offerings include Google App Engine and AWS Elastic Beanstalk.

Infrastructure as a Service (IaaS). In this model, users receive a complete computing environment and have control over the operating systems and applications deployed [7, p. 174]. The cloud provider remains responsible for managing the virtual and physical infrastructure, including networking, storage, servers, and virtualization [6, p. 4]. The primary advantage of IaaS is its pay-as-you-go pricing model, along with the ease of scaling resources as demand grows [9, p. 6]. Examples of IaaS offerings include AWS EC2 and Google Compute Engine.

The use of cloud service models offers many advantages for users. One of the main benefits of using a cloud service instead of building an in-house IT infrastructure is the reduction in costs. Cloud computing typically does not require the purchase of expensive hardware; the only cost incurred is based on actual resource usage. In addition to cost-efficiency, cloud computing offers significant advantages in scalability and flexibility. Resources can be easily scaled up or down depending on demand, whereas this process would be time-consuming and expensive for a company managing its own infrastructure [8, 7].

However, cloud computing also presents some disadvantages. The primary concern is data security [6, p. 3]. Cloud services commonly follow a multi-tenancy model, meaning that several users share the same physical infrastructure or software [8, p. 11]. Furthermore, the exact physical location of stored data is often unknown to the users, raising concerns about confidentiality, access control, and trust.

Numerous security issues and threats associated with cloud computing are discussed extensively in cloud security research. Singh et al. (2017) provide a comprehensive overview of these concerns and related studies [12]. Some of these security concerns can be addressed through the use of different cloud deployment models. Cloud computing is typically categorized into four deployment models:

Public Cloud. The public cloud is the most common form of cloud computing. It refers to publicly available resources that can be provisioned via self-service over the Internet [9, p. 6]. Public clouds are managed and operated by cloud providers—typically businesses, but they can also be academic or governmental organizations. The physical infrastructure resides on the provider's premises [11, p. 3].

Private Cloud. Access to a private cloud is limited to a single organization, although it may serve multiple users within that organization. Private clouds can be managed either

by the organization itself or by a third-party cloud provider [11, p. 3]. Since they can be deployed within the organization's own network, private clouds can offer enhanced security [6, p. 4].

Community Cloud. Community clouds are similar to private clouds, with the key difference being that they are used by multiple organizations. The infrastructure can be owned and managed by one or more of the participating organizations or by a third-party provider [11, p. 3]. Organizations sharing a community cloud typically have common interests, goals, or concerns [6, 11].

Hybrid Cloud. A hybrid cloud is a combination of two or more of the aforementioned deployment models. While the individual clouds remain distinct, they are interconnected through standardized or proprietary technologies that enable the interoperability of data and applications [11, p. 3].
Cloud computing enables a wide range of application areas. One of these is big data analytics, which—as the name implies—deals with processing and analyzing large volumes of data. Cloud platforms offer the computational power and storage needed for such tasks [6, p. 4f].

The evolution of cloud technologies has also influenced software development and testing. Developers can now use cloud-based development tools and platforms, facilitating remote collaboration and continuous integration. Another area that benefits from cloud computing is the Internet of Things (IoT). In IoT systems, everyday objects are augmented with sensors and computing capabilities. Cloud services can provide the additional storage and processing power required to support these devices.

Gaming is another area likely to be significantly impacted by cloud computing in the future. By running games on cloud infrastructure, users are not required to install them locally or purchase high-end hardware. However, this model is only feasible with reliable and fast network connections [6, p. 4f]. Some existing services utilizing this model include Google Stadia, PlayStation Now, and Xbox Game Pass.

Due to the broad adoption of cloud platforms across diverse application domains, there is a growing need to develop optimized solutions for challenges such as placement, scheduling, energy efficiency, and virtual machine (VM) migration policies [13, p. 19923]. To assess the performance and effectiveness of these solutions, thorough testing is required.

One way to test these solutions is by creating physical testbeds using actual cloud infrastructure. However, this approach is often impractical due to high costs and time requirements. Therefore, the testing of cloud environments is commonly conducted through simulation [13, p. 19923].

The remainder of this thesis is structured as follows: Section 2 discusses the current state of cloud computing, beginning with enabling technologies such as virtualization and containerization. It also provides a brief overview of the cloud services market by comparing major providers and concludes by outlining the theoretical foundations of cloud simulation.

Section 3 introduces the research questions and outlines the objectives of this study, along with the expected contributions to the field of cloud computing research.

Section 4 specifies the functional and non-functional requirements for the implementation and details the rationale behind key architectural and design decisions.

The core contribution of this thesis, modeling spot instance behavior and extending a cloud simulation framework, is presented in Section 5.

Section 6 introduces a virtual machine allocation algorithm from existing literature and presents an adjusted version designed to reduce the number of spot instance interruptions.

Finally, Section 7 provides a systematic evaluation of the developed solution. It includes a usage example of the extension, the simulation of spot instances using real-world trace data, and a performance comparison of the implemented allocation algorithms.

2 State of the Art

This section presents the state of the art in cloud computing technologies. It begins with a discussion of current technological trends that facilitate the adoption and operation of cloud computing environments. The second part provides an overview of the current cloud market, focusing on the services offered by the leading cloud providers.

Given the focus of this thesis on the simulation of virtual machine instances, the market analysis and technological discussion will primarily center on services aligned with the IaaS model. While SaaS currently accounts for the largest share of cloud service revenue in terms of end-user spending, IaaS is projected to be the fastest-growing service model, according to forecasts published by Gartner [14].

2.1 Current Technology Trends

This subsection explores key technologies that enable Infrastructure as a Service offerings in cloud computing. The foundational technology is virtualization, which allows multiple isolated environments to run concurrently on a single physical host. More recently, containerization has emerged as an alternative with different trade-offs. After discussing and comparing virtualization and containerization, common instance purchasing models offered by cloud providers are briefly examined.

2.1.1 Virtualization

Virtualization is a fundamental enabling technology for IaaS cloud environments and has played a pivotal role in the proliferation of cloud computing [15, p. 68]. It allows cloud providers to run multiple isolated virtual machines with different operating systems and applications on the same physical hardware [16]. These virtual machines share underlying resources such as RAM, storage, and CPU [15, p. 69].

Virtualization is implemented through a Virtual Machine Monitor, commonly known as a hypervisor. There are two primary types of hypervisors, illustrated in Figure 2.1.

- **Type 1 Hypervisors:** Also referred to as bare-metal hypervisors, these run directly on the host hardware without requiring a host operating system [17, p. 3]. This allows for direct access to hardware resources, resulting in improved performance and security. Type 1 hypervisors typically offer higher VM density due to lower processing overhead [18, p. 23f]. Examples include Xen, Microsoft Hyper-V, and VMware ESXi.
- **Type 2 Hypervisors:** Also known as hosted hypervisors, these operate on top of a conventional host operating system. Virtual machines are executed as processes

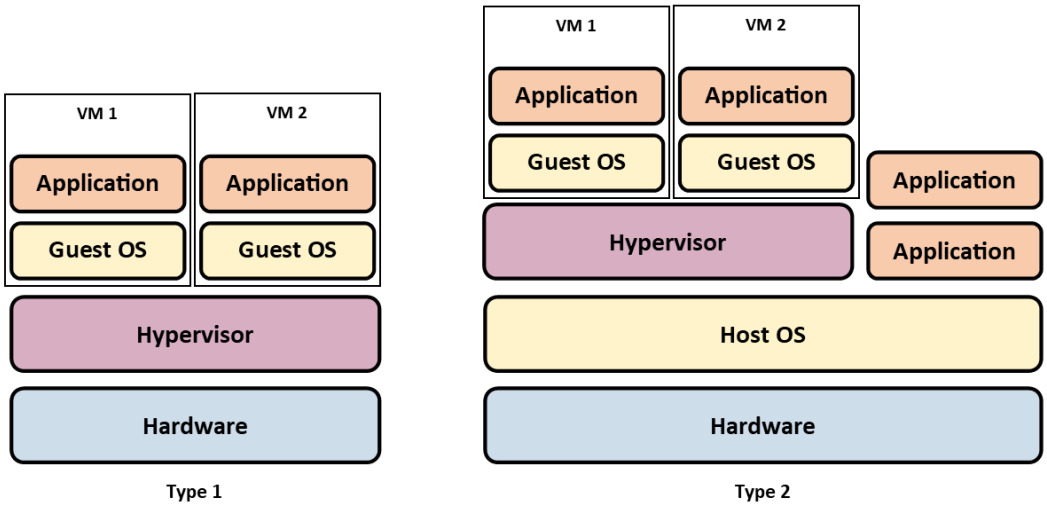

Figure 2.1: *Hypervisor Architecture Types [1]*

within this OS [15, p. 69], which manages hardware access on behalf of the hypervisor [18, p. 25]. Type 2 hypervisors are generally easier to install and use but introduce additional overhead and potential points of failure. Common examples include VMware Workstation and Oracle VirtualBox.

A robust hypervisor must ensure a high degree of isolation between VMs, manage the allocation and sharing of physical resources transparently, and support administrative functions such as VM creation, monitoring, and maintenance [17, 18, p. 5]. Resources must be provisioned in a way that preserves the illusion of exclusive ownership, preventing interference between co-located VMs.

There are various types of virtualization, among which *hardware virtualization*, also known as *server virtualization*, is the most prevalent in cloud computing [19, p. 1133]. This method virtualizes physical hardware, allowing multiple guest operating systems to run as if they were installed directly on physical machines [20, p. 37]. Server virtualization is typically divided into the following subtypes [19, p. 1133]:

- **Full Virtualization:** In this model, an unmodified operating system runs as the guest OS. The guest is unaware of being virtualized and operates identically to a non-virtualized system [21, p. 51]. Full virtualization involves complete emulation of the underlying hardware, ensuring strong isolation between host and guest systems.
- **Paravirtualization:** In contrast, paravirtualization requires a modified guest OS that is aware of its virtualized environment. This model typically offers better performance due to reduced overhead but limits OS compatibility and portability. It is most suitable for open-source operating systems where source code can be modified [19, p. 1133].

Another frequently used type of virtualization is *Client or Desktop Virtualization*. It enables users to run multiple operating systems on a single device [20, p. 37]. There are two main types of desktop virtualization:

- **Virtual Desktop Infrastructure (VDI):** In VDI, multiple virtual desktops run in virtual machines on a centralized server. These desktops can be accessed from any device [22]. A key advantage is centralized management of desktop images rather than relying on individual user devices. Thin clients—low-power, inexpensive, and reliable devices—are typically used to access these desktops [18, p. 17f]. Since data resides on the server, there is minimal risk of data loss if the thin client device is lost, damaged, or stolen [18, p. 18]. This approach offers users flexibility, as they can access their desktop environments from various devices and are not restricted to a single location [19, p. 1135].
- **Local Desktop Virtualization:** In this model, the hypervisor is installed directly on a local device, allowing users to run additional operating systems that are independent of the host OS [22].

Storage Virtualization separates logical from physical storage through abstraction [19, p. 1134]. It aggregates multiple storage devices across a network to appear as a single storage system [22], improving storage efficiency, simplifying updates, and enhancing availability and utilization. Storage virtualization can be implemented using Direct Attached Storage (DAS), Storage Area Network (SAN), or Network Attached Storage (NAS) [19, p. 1134].

Application Virtualization allows applications to run on devices without being installed on the underlying operating system [20, p. 37]. The main approaches include:

- **Server-based Application Virtualization:** Applications are executed entirely on the server; users interact only with the graphical interface [22].
- **Local Application Virtualization:** The entire application is executed on the local device [22].
- **Application Streaming:** A hybrid approach where the application runs on the server but sends small components to the client for execution [22].

Network Virtualization aggregates hardware and software network resources into a single virtual network [20, p. 37]. It is typically categorized into Network Function Virtualization (NFV) and Software-Defined Networking (SDN) [23, p. 2]. While their purposes differ, they are complementary. NFV virtualizes hardware-based network functions (e.g., firewalls and load balancers), enhancing manageability. SDNs virtualize the controllers in connectivity devices, enabling centralized management of packet flow and policies, which improves resource optimization [23, p. 2].

Memory Virtualization abstracts memory resources across multiple devices and presents them as a unified memory pool. It can be implemented at either the application level, where applications access shared memory directly, or at the operating system level, where the OS provides access to virtualized memory [19, p. 1134].

The final virtualization type discussed here is *OS-Level Virtualization*. In this model, virtual instances (containers) are executed on top of an existing host OS without their own OS kernel. Instead, they use shared system libraries [20, p. 37]. OS-level virtualization—commonly referred to as container-based virtualization—is further explored in Section 2.1.2.

Virtual Machines. VMs provide users with access to virtualized computing resources. A VM typically consists of virtual disk files and a configuration file [18, p. 37ff]. The configuration file defines which physical resources (CPU, memory, storage) the VM can utilize. The hypervisor provisions these resources via standardized virtual devices, allowing for portability between different virtualization platforms [18, p. 37ff]. VMs emulate physical computers and are operated in a similar fashion.

2.1.2 Containerization

Container-based virtualization is a lightweight alternative to traditional hypervisor-based virtualization [2, p. 1ff]. It employs OS-level virtualization, leveraging kernel features to manage CPU, memory, and network resources while executing isolated processes [24, p. 22]. The container architecture is illustrated in Figure 2.2.

A major drawback of hypervisors is that each VM requires its own OS and libraries, resulting in large image sizes and significant performance overhead. Containers, by contrast, are more lightweight and better suited for high-performance computing (HPC) environments. They consume fewer resources, start faster, and are more energy-efficient [24, p. 22]. For instance, container startup times can be as low as 50 milliseconds, while virtual machines often take 30 to 40 seconds [2, p. 4]. Bhardwaj et al. (2021) reviewed several studies comparing hypervisor and container performance. Their findings show:

- For CPU-intensive tasks: containers (Docker, LXC, rkt) had 2.5–4% overhead, while KVM had 42%.
- For memory-intensive tasks: containers had 0.97–3.11% overhead, while KVM had 14.98%.
- For network and disk-intensive tasks: containers had 6.7–17.81% overhead, while KVM had 48.29% [2, p. 5ff].

However, containers have some limitations. Unlike VMs, they share the host operating system, which limits OS flexibility and may pose security concerns [2, p. 4]. The widespread adoption of containerization and container clusters introduces challenges such as IP management, cluster elasticity, and workload rescheduling [2, p. 12]. To address these issues, container orchestration platforms like Docker Swarm and Kubernetes have emerged. While Docker Swarm supports only Docker containers, Kubernetes supports various container runtimes [2, p. 12]. Other orchestration tools include Apache Mesos, Marathon, YARN, and Omega [24, p. 23].

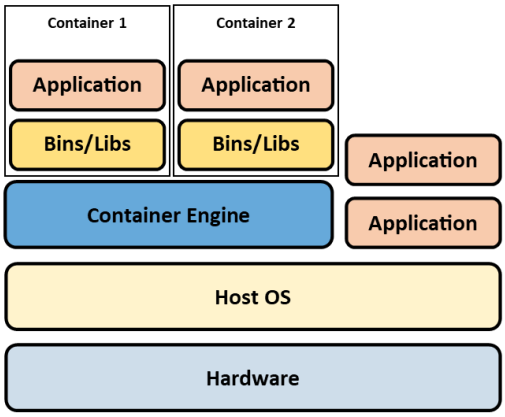

Figure 2.2: *Containerization architecture [2]*

2.1.3 Common Purchase Models

Virtual machine instances are generally offered through three widely adopted purchase models. While different cloud providers may use varying terminology, this section follows the terminology used by Amazon Web Services, the market leader [10]. These models are *On-Demand Instances*, *Reserved Instances*, and *Spot Instances*.

On-Demand Instances. On-demand instances represent the standard and most flexible model for purchasing virtual machine instances from cloud providers [25]. This model operates on a *pay-as-you-go* basis, meaning that users are not required to make long-term commitments or upfront payments. Payment is only required for the actual time the instance is in use [26, p. 3].

Users have full control over the instance lifecycle, allowing them to start and stop instances at will [25]. Most major cloud providers offer *per-second billing* with a minimum usage duration of 60 seconds [27, 25]. Due to their flexibility, on-demand instances are ideal for applications that must not be interrupted and have unpredictable or short-term spikes in resource demand [28].

Reserved Instances. Reserved instances require users to commit to a longer usage period, typically one or three years [25, 28, 27]. They are suitable for workloads with predictable, long-term resource demands. However, predicting future resource needs accurately can be challenging, and incorrect forecasts may lead to underutilization and financial losses [29, p. 1].

Cloud providers encourage the adoption of reserved instances by offering substantial discounts—up to 72% compared to on-demand pricing, depending on the instance type. Payment options typically include full upfront payment, monthly installments, or partial upfront payments followed by monthly charges [26, 25, p. 4]. If a reserved instance no longer meets the user's requirements, some providers allow modifications. These

modifications may include changing the availability zone, instance size, or scope, or even splitting a reservation into smaller ones [25, 28, 27].

However, the cancellation of reserved instances is generally not permitted. Once purchased, users are obligated to pay for the reserved term, whether or not the instance is used [25, 28, 27]. One notable exception is Microsoft Azure, which allows early termination of reserved instances. In this case, the remaining cost is refunded, though an early termination fee may apply [28].

AWS provides an *instance marketplace*, allowing users to resell unused reserved instances. All payment options are supported for resale, but only the upfront portion is covered in the sale. Instances paid for in monthly installments are often listed at $0. Sellers can set their own prices, though market value recommendations are provided [29, p. 6]. AWS charges a *12% service fee* for successful transactions [30, p. 298]. Until the instance is sold, it remains under the seller's ownership and can still be used [25]. Researchers such as Yang et al. (2019) have proposed algorithms to help users determine optimal selling strategies for these instances [30, p. 303].

Spot / Preemptible Instances. Spot or preemptible instances offer significantly discounted prices by utilizing unused capacity in cloud data centers [25, 31]. Discounts can reach up to 90% compared to on-demand prices. However, this model has a critical limitation: the provider can interrupt and reclaim the instance at any time if capacity is needed elsewhere or if the user's bid price falls below the market rate [25].

Initially, spot pricing was determined purely by supply and demand, resulting in *highly volatile prices* [32, p. 19]. Consequently, much early research focused on developing bidding strategies. In 2017, AWS changed its pricing model to reduce volatility. The new model appears to be based on long-term trends in supply and demand, but the exact algorithm remains undisclosed [32, p. 19].

Although the changes led to more stable and lower average prices, this was primarily due to the removal of extreme price spikes. George et al. (2019) found that when these spikes are excluded, prices in the new model were generally higher [31, p. 5]. Therefore, while spot instances have become cheaper for long-term usage, *short-term use cases have become comparatively more expensive* [32, p. 25]. AWS provides historical pricing data to help users monitor price trends [33].

Spot instances are best suited for workloads that are *fault-tolerant* and *not time-sensitive* [28, 27]. Their operational lifecycle and behaviors are discussed further in Section 4.1.3.

Dedicated Hosts. Cloud environments typically use *multi-tenancy*, where virtual instances from multiple users share the same physical server. In contrast, *dedicated hosts* offer users exclusive access to an entire physical server. Although not a purchase model in themselves, dedicated hosts can be used with both on-demand and reserved instance pricing models [25, 28]. This option is often chosen to meet regulatory, licensing, or performance isolation requirements.

2.2 Major Cloud Platforms and Services

In this section, the major players in the cloud infrastructure market are reviewed. The providers selected for this analysis are AWS, Microsoft Azure, and Google Cloud, as they hold the largest market shares in the cloud computing sector [10].

For this review, accounts were created with each provider, and information was gathered from official documentation, service portals, and product pages. The analysis is structured as follows: first, the basic functionality of each provider's virtual machine services is examined. Then, pricing is compared using two similarly configured instance types across all providers and regions to maintain consistency for the evaluation in Section 2.2.

Subsequently, the scalability of each provider is evaluated by analyzing the maximum specifications available for a single virtual machine instance. Additionally, the service-level agreements (SLAs) are reviewed to determine the expected availability of each provider's services. Finally, notable additional features and value-added services are discussed.

Amazon Web Services (AWS)

Amazon Web Services, a subsidiary of Amazon, is a U.S.-based cloud provider and the current market leader with approximately 33% market share [10]. The primary service considered in this analysis is Amazon EC2 (Elastic Compute Cloud), AWS's virtual machine offering [25].

To launch an EC2 instance, users must deploy an Amazon Machine Image (AMI), which contains the necessary configuration to run the instance [25]. AWS provides several predefined AMIs, and users can also create custom AMIs or use shared ones available through the AWS Marketplace. An AMI is not restricted to a single instance and can be reused across multiple launches.

To deploy an AMI, a few requirements must be satisfied [33]. AWS supports a range of operating systems, including multiple Linux distributions and Windows Server. The supported AMI architectures include 32-bit (i386), 64-bit (x86_64), and 64-bit ARM (arm64) [33].

Scalability. AWS offers a wide range of instance types. The smallest instance, `t2.nano`, includes 1 vCPU and 512 MiB of memory, without dedicated storage and with low to moderate network performance. On the high end, the `x1e.32xlarge` instance provides 128 vCPUs, 3,997,696 MiB of memory, 3840 GB of SSD storage, and up to 25 Gbps of network performance. The largest configurations available offer up to 60,000 GB of storage and 100 Gbps network throughput [33].

Availability. The availability of AWS services is defined in its Service Level Agreement [34]. AWS guarantees a monthly uptime of 99.99% for EC2 instances. If availability falls below this threshold, customers are eligible for service credits toward future usage. The value of these credits increases with the severity of the outage. If monthly uptime drops below 95%, AWS must refund 100% of the affected charges as credits [34].

To help users understand the geographic distribution of their services, AWS divides its infrastructure into multiple global regions. Each region is fully independent and includes multiple *Availability Zones*—isolated yet interconnected data center clusters [25]. AWS maintains multiple regions across Europe, North America, and Asia Pacific, and also operates one region each in South America, the Middle East, and Africa [25].

Additional Features. Upon account creation, AWS provides a selection of free tier offerings [35]. These are divided into services that are always free and those that are free for the first 12 months. The free tier includes, among other benefits, 750 hours of EC2 `t2.micro` usage per month and 25 GB of NoSQL database storage.

For students and educators, AWS offers specialized programs such as *AWS Educate* and *AWS Academy*, which provide free learning resources and access to cloud tools [35].

To enhance security, AWS uses *Security Groups*, which act as virtual firewalls controlling traffic to and from EC2 instances [25]. Users can define inbound rules to restrict traffic based on source IP addresses, protocols, and ports. Only traffic matching the defined rules is permitted to reach the instance.

Microsoft Azure

Microsoft's cloud platform, Azure, is the second-largest cloud infrastructure provider, holding approximately 18% of the global market share [10]. As with the other providers, this analysis focuses on Azure's virtual machine services.

Azure offers seven main virtual machine families, each targeting different use cases: General Purpose, GPU, High Performance Compute, Compute Optimized, Memory Optimized, Storage Optimized, and Confidential Compute [36]. In the Europe West region alone, users can select from over 290 instance types, varying in CPU and memory configurations. The selected deployment region may limit the available instance types.

Customers can choose from predefined Linux and Windows Server operating systems. Alternatively, custom machine images and those available through the Azure Marketplace are supported [36].

Scalability. To assess scalability, the virtual machine types with the highest specifications were reviewed. The largest CPU configuration is available in the "HB120rs v2" instance, a High Performance Compute machine, which provides 120 vCPUs and 468.75 GB of memory [36]. The instance with the highest memory, the "E96a_v4," offers 672 GB of memory and 96 vCPUs. This instance is part of the Memory Optimized family [36].

Availability. Azure's SLA for virtual machines specifies different availability guarantees based on the deployment configuration [37]. Microsoft defines three scenarios:

- Instances deployed across multiple regions: 99.99% monthly uptime
- Instances deployed two or more times within a region or zone: 99.95% uptime

- Single-instance deployments: 99.9% uptime

If these guarantees are not met, users can request service credits for future billing cycles. The credit amount depends on the actual measured uptime. For uptime below 95%, users are entitled to 100% credit for the impacted service [37].

Azure's infrastructure is distributed across numerous regions worldwide. Multiple regions are located in Europe, North America, Asia, and Australia, with additional regions in South America, the Middle East, and India [36]. Each region contains several availability zones, which are physically separate data centers within the region [28].

Google Cloud

Google Cloud is the third-largest cloud infrastructure provider with an estimated market share of 8% [10]. The service analyzed here is the *Google Compute Engine*, which provides virtual machine instances hosted on Google's infrastructure.

To deploy a Compute Engine instance, users must specify the deployment region and zone, as well as the machine type [38]. Google Cloud offers several general-purpose machine types (N1, E2, N2, N2D), along with memory-optimized (M1) and compute-optimized (C2) types [27]. These types differ in terms of vCPU-to-memory ratios and underlying CPU platforms. For N1 instances, up to 8 GPUs can also be attached [38]. Instances can be purchased as on-demand, committed-use, or preemptible (spot) instances.

Google supports a variety of Linux distributions and Windows Server as operating systems. Users may also upload custom machine images. While there are no strict limitations on custom images, certain features are only available when using OS versions specifically optimized for Google Cloud [27]. Images from the Google Cloud Marketplace can also be used [38].

Scalability. Google Cloud offers a range of scalable instance types. Among general-purpose instances, the `n1-ultramem-160` and `m1-ultramem-160` provide the highest available resources—up to 160 vCPUs and 3.75 TB of memory [38]. Users can also define custom machine types with up to 96 vCPUs and 624 GB of memory [38].

Google's Tensor Processing Units (TPUs), described in Section 2.2, can scale up to 2048 cores, though availability is restricted to selected regions and pricing is available only on request [27].

Availability. The SLA for Google Compute Engine outlines the availability guarantees for different configurations [39]:

- Multi-zone deployments: 99.99% monthly uptime
- Single-instance deployments: 95.5% monthly uptime

If these levels are not maintained, customers can request *Financial Credits*, which reduce future billing charges. For example, a multi-zone instance with less than 95%

uptime or a single instance with less than 90% uptime entitles the customer to a credit equal to 50% of the relevant charges for that month [39].

Google Cloud maintains multiple regions across Europe, North America, and Asia, and one region each in Australia and South America. Each region is divided into zones, which represent isolated clusters of physical infrastructure. Available services and instance types can vary between regions and zones [27].

Additional Features. Google Cloud offers a free tier for new accounts [27]. New users receive $300 in credits valid for 12 months. These free-tier quotas are more limited than the default quotas described earlier. Users may upgrade to a paid account to make the credit permanent and unlock higher quotas, but they may incur charges if usage exceeds the available credit.

Additionally, some services are always free, such as a single `f1-micro` virtual machine instance, limited by monthly usage hours [27].

A notable feature of Google Cloud is its support for *Tensor Processing Units (TPUs)* [27]. Developed by Google, TPUs are specialized hardware accelerators optimized for machine learning workloads, particularly those using the TensorFlow framework. TPUs are highly efficient at executing vector and matrix computations and are commonly used to train complex neural networks.

Overview

This section presents a side-by-side overview of the results from the market analysis for AWS, Microsoft Azure, and Google Cloud.

Basic Functionality. In terms of basic functionality, AWS, Azure, and Google Cloud offer similar capabilities. All three providers allow deployment of virtual machine instances across multiple global regions and offer predefined host operating systems, including various Linux distributions and Windows Server.

Azure provides the widest variety of instance families, whereas AWS offers the largest number of individual instance types. Google Cloud distinguishes itself by allowing users to define custom virtual machines, specifying an arbitrary combination of vCPUs and memory—something not currently supported by AWS or Azure.

All three platforms support on-demand, reserved, and spot instance purchase models. However, Google Cloud's spot instance equivalent, referred to as *preemptible virtual machines*, is more limited in functionality. These instances have a maximum lifetime of 24 hours and cannot automatically relaunch after termination [27].

Scalability. Scalability was assessed based on the maximum specifications available for individual virtual machine instances. In terms of individual instance capacity, AWS provides the instance with the highest memory (4192 GB and 128 vCPUs), while Google Cloud offers the instance with the most vCPUs (160 vCPUs and 3750 GB of memory).

Azure lags behind slightly, with a maximum configuration of 120 vCPUs and 672 GB of memory.

Additionally, Google Cloud offers Tensor Processing Units (TPUs), which provide up to 2048 vCPUs and 32 GB of memory, although their availability is restricted to selected regions.

Availability. All three cloud providers offer similar service availability guarantees, though there are subtle differences. AWS guarantees a monthly uptime of 99.99% for all EC2 compute services [34]. Azure offers 99.99% only for instances distributed across multiple regions, while Google Cloud offers the same for instances deployed across multiple zones (see Sections 2.2 and 2.2).

The policies on service credits also vary. AWS offers credits of 10%, 30%, or 100% based on the actual uptime. Azure provides credits of 10%, 25%, or 100%, while Google Cloud only offers up to 50% of the affected charges. Based on these conditions, AWS provides the most favorable availability terms overall [34, 37, 39].

Free Tier Availability. All three providers offer free tier programs to attract new customers, though the details differ in terms of scope and duration. Each provider includes both limited-time and always-free resource tiers.

Azure and Google Cloud offer credits in addition to the free tier: $170 for 30 days on Azure and $300 for 12 months on Google Cloud. Google Cloud stands out by allowing users to convert to a paid account and use their remaining credits without restrictions, whereas AWS focuses more on predefined free usage limits.

In addition to general free tier offers, all three platforms provide educational programs and additional resources for students, educators, and academic institutions.

Pricing. To compare pricing, two instance types were selected for each provider: a low-cost, general-purpose instance with 2 vCPUs, 8 GB of memory, and 16 GB of storage; and a medium-cost, general-purpose instance with 32 vCPUs and 128 GB of memory. The prices, shown in Tables 2.1 and 2.2, reflect hourly on-demand rates for Linux instances, excluding additional services or features [33, 36, 38]. They were obtained from the web console of each respective cloud provider and therefore represent provider-supplied estimates. For all providers, the Frankfurt region was chosen for the comparison: “Germany West Central” for Azure, “europe-west3” for Google Cloud, and “eu-central-1” for AWS.

In this comparison, Google Cloud offers the lowest on-demand price for the small instance type. For the large instance type, Azure and AWS provide identical on-demand pricing. However, Azure stands out as the most cost-effective option for both reserved and spot (or preemptible) instances across these instance types.

Conducting experiments on live cloud infrastructure often incurs high costs and introduces significant operational complexity. As a result, simulation offers a practical and scalable alternative for evaluating cloud-based systems and strategies. The following section presents the state of the art in cloud simulation, including an overview of widely used simulation frameworks.

Table 2.1: *Pricing Comparison for Small Cloud Instances (July 2025)*

Provider	Instance Type	On-Demand	Reserved / Spot Price
AWS	t2.large	$0.1072	$0.07639 / $0.04287
Google Cloud	e2-standard-2	$0.0889	$0.05702 / $0.03715
Azure	B2ms	$0.0960	$0.056 / $0.02098

Note: Prices are in USD per hour. Reserved prices reflect 1-year commitment rates where applicable.

Table 2.2: *Pricing Comparison for Large Cloud Instances (July 2025)*

Provider	Instance Type	On-Demand	Reserved / Spot Price
AWS	m5.8xlarge	$1.840	$1.159 / $0.552
Google Cloud	n2-standard-32	$2.0046	$1.2626 / $0.5812
Azure	D32\v5	$1.840	$1.086 / $0.336

Note: Prices are in USD per hour. Reserved prices reflect 1-year commitment rates where applicable.

2.3 Cloud Simulation

Cloud platforms are growing in scale and complexity, making experimentation on real infrastructure both costly and time-consuming [40, p. 651]. In addition, cloud environments vary significantly in hardware, software, and networking configurations, which further complicates the design and replication of experiments [4, p. 24]. These factors limit the feasibility of conducting large-scale, repeatable testing directly on real cloud infrastructures [40, p. 651]. As an alternative, simulation provides a scalable, cost-effective method to estimate the performance and behavior of cloud environments [4, p. 24]. A number of frameworks and tools exist that support modeling and simulating cloud environments; these are discussed in Section 2.3.1.

Advantages of Simulation. Simulation offers several advantages over physical experimentation. It requires significantly less time and effort to implement and provides greater flexibility and applicability [4, p. 24]. Compared to real-world testing, simulation allows users to exert greater control over both the environment and the experimental conditions [3, p. 70]. This makes it easier to reproduce results and eliminates interference from unpredictable real-world events. Additionally, simulation can often be executed much faster than real systems, enabling longer periods to be analyzed within a shorter timeframe [3, p. 70]. Parameter studies are also more efficient, as multiple simulation runs with varying inputs can be performed in parallel.

Disadvantages of Simulation. Despite these benefits, simulation also presents several limitations. Models rely on simplifications and assumptions that may not fully reflect the behavior of real-world cloud systems [40, p. 652]. Inaccurate or overly simplistic models can result in misleading results. Furthermore, developing high-fidelity simulation models can be both time-consuming and resource-intensive [3, p. 71]. Therefore, it is essential to assess whether simulation is the appropriate method for a given scenario or if physical experimentation is more suitable [40, p. 652].

Simulation Modeling. Simulation modeling involves creating abstractions of real-world systems to describe or replicate their behavior [41, p. 54f]. The scope of a simulation model must be carefully constrained to ensure it captures only the essential aspects of the system. Increased model complexity does not guarantee increased accuracy; instead, a model's quality is determined by how well its outputs match real-world results [41, p. 55].

Discrete Event Simulation. Most cloud simulation tools employ *Discrete Event Simulation (DES)* as their underlying methodology [40, p. 651]. DES models the evolution of a system over time by changing state variables only at discrete points—when specific events occur [3]. At each event, entities within the simulation (such as virtual machines or hosts) compete for limited system resources [42, p. 736f].

DES models typically consist of five core components: *entities*, *events*, *resources*, *operations*, and *control elements*.

- **Entities** represent the dynamic components of the system. In a cloud context, these could include virtual machines, tasks, or hosts. Entities may be internal (generated by the simulation itself) or external (defined by the modeler). They may exist in various states such as active, ready, time-delayed, condition-delayed, or dormant.

- **Events** are instantaneous occurrences that change the state of the system. These can be triggered by entities or by the system itself. Examples include the creation or termination of virtual machines.

- **Resources** are the system components needed to carry out operations—such as CPU, memory, or storage. These are typically limited, requiring entities to compete for access.

- **Operations** are actions executed by entities during the simulation (e.g., processing a cloudlet).

- **Control Elements** help to manage simulation logic. These may include conditions or user-defined rules that guide entity behavior.

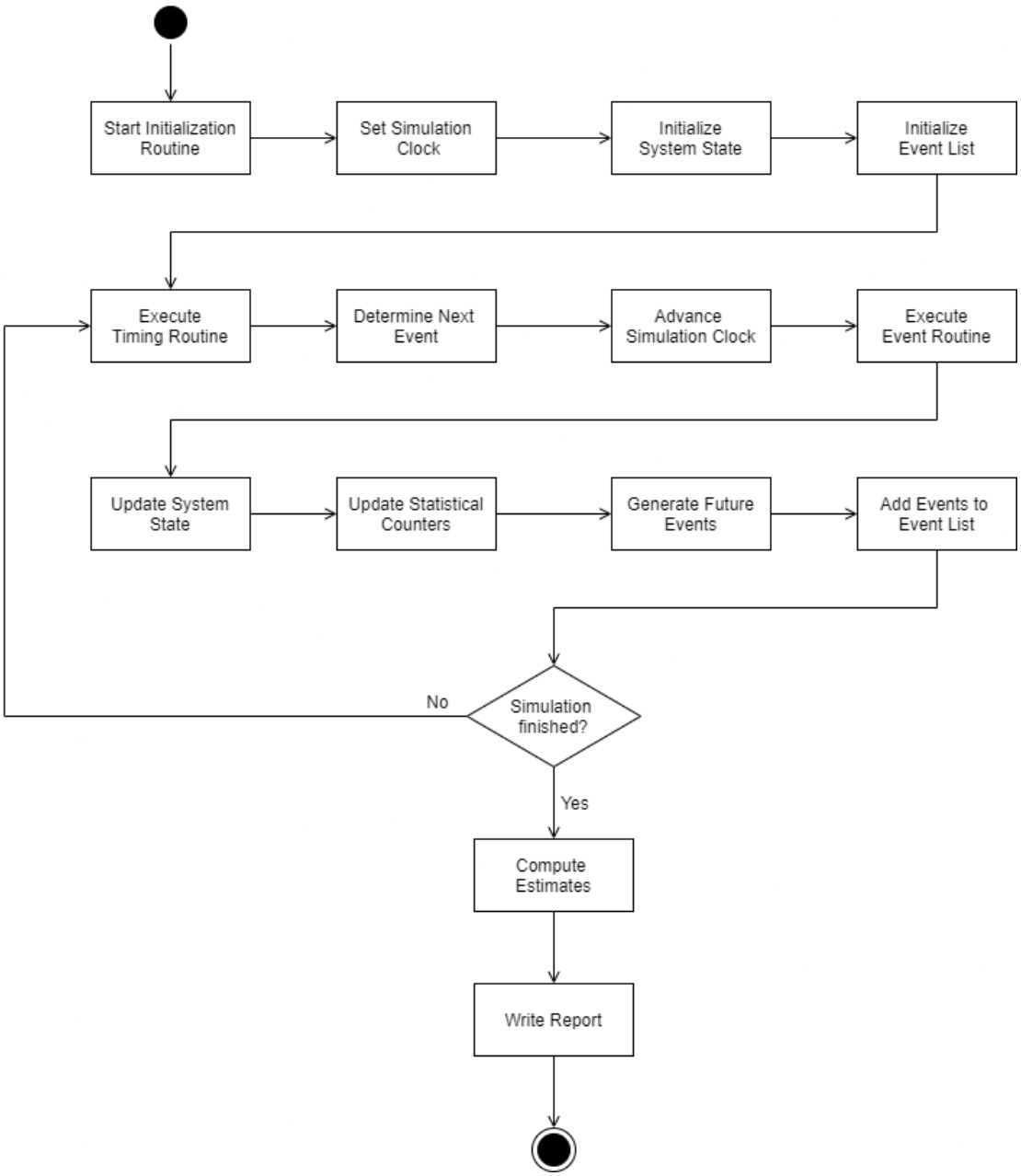

Figure 2.3: *Generalized execution of a Discrete Event Simulation [3]*

Event Scheduling in DES. DES uses a simulation clock to manage time progression [42, p. 737]. At the beginning of a simulation, the clock is initialized to zero, and all events are assigned time stamps. Two major approaches exist for advancing time:

- **Next-event time advance**: The clock jumps directly to the time of the next scheduled event. The state of the system is updated accordingly, and the process

continues until a stopping condition is met or no future events remain [3, p. 8f].

- **Fixed-increment time advance**: The clock advances in fixed time steps. All events that occur within each interval are processed at the end of that interval. This approach is suitable when events align with regular time patterns (e.g., hourly, weekly) [3, p. 72f].

DES Workflow. Figure 2.3 illustrates a typical DES workflow using the next-event time advance method. Initially, the simulation clock and system state variables are initialized, including event lists (e.g., future, delayed). The simulation then enters a loop:

1. The timing routine selects the next event and updates the clock.
2. The event routine updates the system state and statistical counters, and may generate new events.
3. The simulation checks whether a stopping condition has been met.

If no termination condition is reached, the loop continues. Upon completion, a report is generated that summarizes the simulation results [3, p. 10].

Other Simulation Methods. While discrete event simulation is the most commonly used approach in cloud simulation tools, several alternative simulation methods exist, each with distinct characteristics and applications.

- **Deterministic Simulation:** In deterministic simulation models, all system components behave in a fully predictable manner. There are no random variables involved, which means that identical input conditions will always yield the same output [3, p. 6].
- **Stochastic Simulation:** In contrast to deterministic models, stochastic simulation incorporates at least one randomized input variable. As a result, each simulation run may produce different outputs. The goal is to estimate the behavior of a system under uncertainty [3, p. 6].
- **Agent-Based Simulation (ABS):** ABS can be seen as an extension of discrete event simulation, in which autonomous agents—representing entities such as users, applications, or infrastructure components—interact with each other and their environment. These agents follow predefined rules and can adapt their behavior during the simulation. Unlike DES, where control is typically centralized, agent-based models emphasize decentralized decision-making [43, p. 136], [3, p. 694].
- **Continuous Simulation:** Unlike discrete simulation, where system state changes occur at specific event times, continuous simulation models evolve continuously over time. State changes are governed by differential equations. Due to the complexity of these equations, analytical solutions are often impractical or impossible

[3]. Continuous simulation is commonly used in engineering and physical system modeling but is less suitable for cloud computing contexts.

- **Hybrid Simulation:** Hybrid simulation combines two or more simulation paradigms—such as discrete event, agent-based, or stochastic models—to better capture the complexity of real-world systems. For example, Bertot et al. (2018) used Monte Carlo methods in conjunction with cloud simulation to improve result accuracy in uncertain environments [44].

2.3.1 Simulation Frameworks and Tools

Bertot et al. (2018) defined two major categories of cloud simulation tools. On the one hand, there are tools that focus on the provider's perspective. These tools are concerned with low-level components of cloud environments, such as operating system kernels and networking. The purpose of tools from this perspective is to assess the design decisions of cloud infrastructures. The other category includes tools that model the entire cloud ecosystem [44, p.3]. To analyze dynamic cloud resources effectively, it is important to also consider client-side behavior. Therefore, the second category will be the focus of this section.

Within this category, several simulation frameworks are available, which differ in their functionality and intended purpose. The most commonly used framework for cloud simulation research is CloudSim. It also serves as the foundation for many other cloud simulation tools and frameworks [40, p. 660]. This section will describe the architecture of simulation frameworks using CloudSim as a basis and briefly discuss additional simulation tools and their applications.

CloudSim. CloudSim is an open-source simulation framework that enables modeling and simulation of cloud computing infrastructures and services. Since CloudSim is the most widely used simulation framework in cloud computing research, its architecture will be described in detail to illustrate the structure and functionality of such frameworks. The primary goal behind CloudSim's development was to support modeling of cloud computing environments, performance testing of application services, and end-to-end simulation of cloud-based network architectures [4, p.25].

CloudSim uses a multi-layered architecture, as illustrated in Figure 2.4.

The first layer, *User Code*, enables users to model cloud scenarios, define requirements, and configure entities such as hosts, virtual machines, and applications [4, p.30]. This layer also includes scheduling policies, allowing users to define custom allocation policies for VMs, applications, and brokers. CloudSim's basic functionality can also be extended at this layer.

The second layer, *CloudSim Simulation*, is responsible for modeling and simulating cloud-based datacenter environments. CloudSim provides a variety of entities, which are instances of component classes that represent cloud models such as datacenters, hosts, and VMs [4, p.31]. Datacenter entities form the base of the simulation and manage host entities that represent the physical machines in the cloud infrastructure. Hosts can be

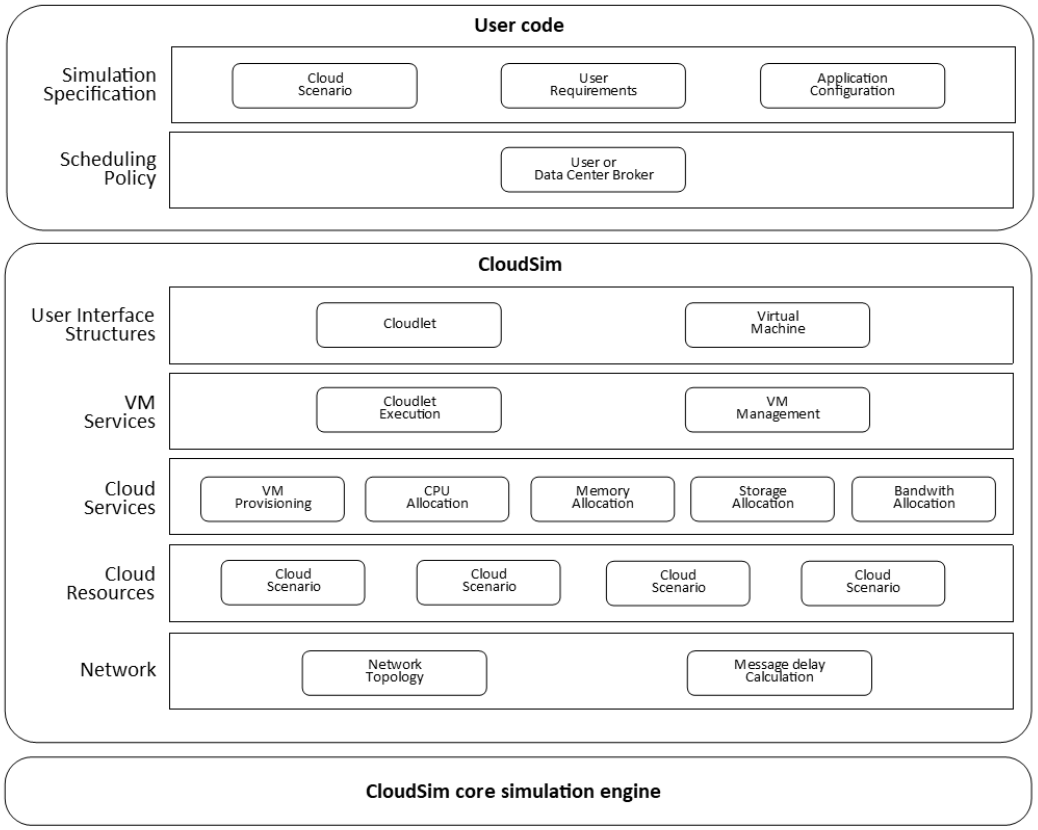

Figure 2.4: *CloudSim - Layered Architecture [4]*

customized for memory, storage, and processing capabilities. VMs are then allocated to hosts based on the defined allocation policy.

To model the execution of applications, CloudSim provides *cloudlet* entities, which are executed on VMs. The scheduling of cloudlets is handled by a datacenter broker. Because cloud service workloads often experience unpredictable demand, entities in CloudSim can be created dynamically at runtime [4, p.31].

CloudSim also models costs to simulate the financial aspects of using cloud resources. Two cost categories are simulated: one related to memory and storage during VM creation, and another related to the execution of applications. In addition to user-incurred costs, CloudSim supports modeling of provider costs such as power consumption, enabling evaluation of energy-aware provisioning algorithms to reduce operational costs, prevent overheating, and lower CO_2 emissions [4, p.36].

The third and final layer, the *Core Simulation Engine*, is a discrete event simulation engine developed specifically for CloudSim. This layer is responsible for managing simulation events and executing the simulation [4, p.39]. The version of CloudSim referenced here is 2.0 [4]. More recent versions have introduced new features such as containerization and improvements to datacenter power models, VM scheduling and allocation strategies [45].

Due to its open-source nature and modular architecture, CloudSim is highly extensible. Consequently, many frameworks have been developed based on CloudSim to improve and expand its features. These include CloudSim Plus (discussed further in chapter 5), as well as WorkflowSim, Cloud2Sim, and iFogSim.

SimGrid. SimGrid is another longstanding and widely used simulation framework. It was originally developed to simulate allocation algorithms in distributed computing environments [46, 47, p.2]. While SimGrid does not natively support cloud-specific features, extensions such as MSG VM API, VMPlaceS, and SchIaaS have added capabilities similar to those found in CloudSim [48].

iCanCloud. iCanCloud is another simulation framework often referenced in comparative studies. Unlike CloudSim and SimGrid, iCanCloud offers a graphical user interface (GUI), which allows users to manage and launch pre-configured VMs, environments, and experiments [49, p.5]. It is also optimized for running large-scale experiments. In comparisons with CloudSim, iCanCloud demonstrated faster execution times for simulations involving a high number of VMs and tasks. However, its memory usage is higher across all experiment scales [49, p.23].

PICS. In contrast to frameworks that simulate the entire cloud environment, PICS focuses exclusively on the concerns of end-users of IaaS clouds [50]. Its goal is to identify the most suitable providers for specific applications and determine which scheduling and resource management policies offer the best cost-efficiency and performance. PICS requires only user-specific configurations and no infrastructure details but is still able to produce accurate results in various scenarios.

Other Simulation Tools. Several additional cloud simulation tools and frameworks exist beyond those discussed here. Due to scope limitations, they are not listed in this thesis. Other studies have provided comprehensive overviews of cloud simulation tools, such as those by Fakhfakh et al. (2017) and Rahman et al. (2019) [51, 13].

2.3.2 VM allocation algorithms

Virtual machine allocation methods can be divided into two categories: static placement and dynamic placement. In static placement, a VM is allocated to a physical host and remains there for its entire lifecycle. Dynamic placement, on the other hand, allows VMs to be reallocated when certain conditions are met or anticipated on the physical host [52, p. 4].

Various objectives define the goals of allocation algorithms, and some of these objectives may conflict. For example, minimizing operational costs and energy consumption must often be balanced with maximizing resource utilization and maintaining service levels as defined by SLAs [52, 53]. In the literature, multiple problem models have been defined to represent different cloud environments. Some models aim to optimize objectives from the cloud provider's perspective, while others explore how customers can distribute their VMs across multiple providers [53, p. 7f].

Bin Packing Problem. VM placement in cloud datacenters resembles the classic bin packing problem [53, p. 14]. In the one-dimensional bin packing problem, a set of items (in this context, VMs) must be placed into N bins (i.e., physical hosts), each with a fixed capacity [54, p. 1061]. VM placement is typically a multidimensional bin packing problem because it must account for multiple resources such as CPU, memory, bandwidth, and storage [55, p. 1].

The main disadvantage of this approach is that it is NP-hard, meaning no polynomial-time solution is known for solving large instances efficiently [52, p. 9]. Common heuristic algorithms used for solving bin packing problems include:

- **First Fit:** Allocates the VM to the first host that meets the resource requirements [56, p. 3].
- **Next Fit:** Similar to First Fit, but skips the first valid host and selects the next one [52].
- **Best Fit:** Chooses the host with the least remaining capacity that still meets the VM's requirements.
- **Worst Fit:** Allocates the VM to the host with the most available resources [57].
- **Random:** Selects a random host until one is found that satisfies the resource demands.

These simple heuristics make placement decisions without optimizing multiple conflicting objectives. Therefore, more sophisticated heuristics and exact algorithms have been

proposed [53]. Since VM placement is NP-hard, exact algorithms provide optimal results but are impractical for large-scale environments due to their exponential runtime [53, p. 15].

VM placement algorithms can be broadly categorized by their optimization goals: minimizing energy consumption, reducing costs, optimizing network traffic, maximizing resource utilization, and enhancing performance [58, p. 3]. Algorithms may optimize multiple objectives (multi-objective) or focus on a single one (mono-objective). The following subsections describe commonly used approaches.

Swarm Intelligence Algorithms (SI)

Swarm intelligence algorithms are inspired by the collective behavior of organisms in nature. Common examples include Grey Wolf Optimization, Ant Colony Optimization, Particle Swarm Optimization, and Whale Optimization.

Grey Wolf Optimization (GWO). GWO is a metaheuristic algorithm inspired by the hierarchical structure and hunting behavior of grey wolves. It supports both mono- and multi-objective optimization [59]. The best candidate solutions are modeled as alpha (best), beta (second-best), and delta (third-best) wolves. A fitness function evaluates solutions over multiple iterations. Candidate positions are updated iteratively, and if better solutions are found, the hierarchy is updated accordingly [55].

Whale Optimization Algorithm (WOA). WOA is based on the bubble-net hunting strategy of humpback whales. A random population of solutions is initialized and evaluated using a fitness function. The algorithm alternates between two phases:

- **Exploration phase:** The position of search agents is updated randomly to explore the search space.
- **Exploitation phase:** Two strategies are used:
  - *Encircling prey:* Updating the position toward the best-known solution.
  - *Spiral updating:* Moving along a helix between the current position and the best-known solution.

A random vector determines whether exploration or exploitation is used in each iteration [60, p. 158–159].

Genetic Algorithms (GA)

Genetic Algorithms are evolutionary algorithms that emulate the process of natural selection [61, p. 319]. Initially, a population of candidate solutions (individuals) is generated [62, p. 139]. Each individual is evaluated using a fitness function, and those with the highest scores are selected for reproduction. Offspring are created through:

- **Crossover:** Combining attributes of two parent individuals.

- **Mutation:** Introducing small, random changes.

The process continues until a stopping condition (e.g., maximum iterations) is reached [62, p. 139]. Finding an appropriate fitness function can be difficult, and GAs are computationally expensive [61, p. 320].

Constraint-Based Weighted Algorithms

Jing et al. (2024) proposed a heuristic algorithm called HLEM-VMP (Host Load Evaluation Model – Virtual Machine Placement), which aims to reduce SLA violations by optimizing VM placement. The approach consists of two steps:

- **Host Filtering:** Candidate hosts must have sufficient available resources to meet VM requirements. The algorithm also penalizes placement of similar workloads to the same host using a CPU-related factor.
- **Host Evaluation:** Candidate hosts are scored using weighted values for each resource type. The host with the best score is selected [63].

Although several cloud simulation frameworks, such as CloudSim and its derivatives, offer extensible architectures for modeling IaaS environments, they typically lack native support for modeling the volatility and interruption dynamics of spot instances. Likewise, existing virtual machine allocation algorithms often assume stable resource availability, limiting their applicability in environments characterized by unpredictable preemptions. These shortcomings indicate a clear need for simulation enhancements that incorporate realistic spot behavior and enable the evaluation of resilient placement strategies. The following section outlines the research questions that this thesis seeks to answer in addressing these challenges.

3 Research Questions

This section presents the primary research objectives and questions addressed in this thesis.

Major cloud providers offer spot or preemptible instances to sell unused capacity at significantly reduced prices. However, these instances can be interrupted at any time, introducing uncertainty and complexity for users. To develop effective cost-saving strategies, it is essential to better understand the behavior of these instances.

Due to the high costs and operational complexity of conducting experiments on live cloud infrastructure, simulation presents a practical and scalable alternative. As such, a major part of this thesis involves developing functionality for simulating spot instance behavior and evaluating the impact of optimized allocation algorithms in such environments.

The central research questions are defined as follows:

ID	Research Question
RQ1	How can spot instances be accurately modeled and simulated using cloud simulation tools?
RQ2	How can Virtual Machine allocation algorithms be optimized to reduce the likelihood of spot instance interruptions through informed host selection in cloud simulation environments?

Table 3.1: *Research Questions*

To address **RQ1**, this thesis investigates how to effectively simulate the key characteristics of spot instances. This includes:

- Modeling spot instance behavior within an established cloud simulation framework.
- Accurately simulating interruption behaviors, including termination and hibernation.
- Extending virtual machine lifecycle states to support dynamic provisioning and resubmission.
- Utilizing real-world trace data as a proof of concept for validating spot behavior modeling and assessing allocation efficiency.
- Identifying factors contributing to interruptions, and examining whether specific instance characteristics can be used to model interruption probabilities.

For **RQ2**, the research is oriented toward enhancing VM allocation strategies. Specifically, it explores how cloud simulation tools can evaluate and improve algorithms under realistic scenarios involving spot instances. Key tasks include:

- Adapting suitable allocation algorithms from existing literature.
- Introducing deterministic modifications to improve the algorithm's ability to handle spot instance interruptions, with the goal of reducing both their frequency and duration.
- Conducting performance comparisons between the improved algorithm and existing allocation methods in scenarios that involve both spot and on-demand instances.

Collectively, these contributions seek to increase the fidelity of cloud simulation environments by integrating realistic resource models commonly utilized by cloud providers. The resulting simulation framework allows researchers and practitioners to rigorously evaluate VM allocation strategies under realistic constraints, providing valuable insights into cost-performance trade-offs within volatile cloud markets.

4 Requirement Analysis

The goal of this section is to identify the features necessary for the implementation. The primary purpose is to improve the understanding of how spot instances behave in a cloud environment, and to derive the corresponding requirements for modeling and implementing this behavior. The section begins with an introduction to the general cloud architecture, followed by a more detailed discussion of the lifecycle and behavior of spot instances. Functional and non-functional requirements based on this background are presented in Section 4.2 and Section 4.3, respectively.

4.1 Background

To accurately simulate the behavior of spot instances within a cloud environment, the first step is to identify the essential entities in the cloud architecture. The National Institute of Standards and Technology defines five major actors in its cloud computing reference architecture: the cloud provider, cloud consumer, cloud broker, cloud auditor, and cloud carrier [64, p. 3].

Cloud Provider. Cloud providers are organizations that offer cloud services and resources to other parties. They are responsible for acquiring and managing the physical infrastructure that underpins cloud environments, including servers, storage systems, and network hardware. In addition, they manage the software stack, such as hypervisors and host operating systems, that enables virtualization and service provisioning [64, p. 7]. A detailed discussion of service models and key market providers has been provided in Sections 1 and 2.2.

Cloud Consumer. Cloud consumers are the users or organizations that acquire cloud resources and services from providers. In an Infrastructure as a Service model, consumers can deploy virtual machines, use storage, and access compute resources to run applications. Pricing is generally based on the quantity and duration of consumed resources. The terms of service delivery are typically defined in Service Level Agreements [64, p. 6].

Cloud Broker. When consumers do not wish to manage or purchase services directly from cloud providers—due to complexity or operational preferences—they may use a cloud broker. Brokers may act as intermediaries offering value-added services, such as identity and access management or enhanced security. Alternatively, they may aggregate services from multiple providers into a unified offering [64, p. 8].

Cloud Auditor. Cloud auditors are responsible for independently assessing cloud services. They evaluate aspects such as security, privacy, and performance. Security audits assess technical and operational safeguards for protecting cloud systems and data. Privacy audits ensure that personal data is managed securely and in compliance with legal and regulatory requirements [64, p. 8].

Cloud Carrier. Cloud carriers facilitate network connectivity between cloud providers and cloud consumers. They typically provide telecommunication and data transport services [64, p. 8].

For the scope of this thesis, the relevant actors for simulating a cloud environment are:

- The *cloud provider*, who offers the infrastructure and resources,
- The *cloud consumer*, who uses and deploys services, and
- The *cloud broker*, who intermediates between providers and consumers.

4.1.1 Cloud Datacenters

The foundation of the cloud infrastructure for service providers lies in their datacenters. Major cloud providers operate multiple datacenters located across various global regions. The regional distribution of datacenters for the major providers was already discussed in Section 2.2, particularly within each provider's availability section.

The physical infrastructure of a datacenter consists of computing resources, storage systems, networking equipment, and supporting infrastructure for power and cooling. Large-scale datacenters operated by leading cloud providers often span multiple buildings, each housing thousands of physical servers [65]. As discussed in Section 2.1.1, virtualization is a critical enabler of cloud computing. Through virtualization, computing power, networking capabilities, and storage are abstracted from the underlying hardware [66, p. 65].

Moreno et al. (2012) proposed a reference architecture for cloud datacenters, in which the core component is the cloud operating system (Cloud OS). The Cloud OS is responsible for managing virtual resources in a multi-tenant environment and must provide several core functionalities to fulfill this role.

Computing resources are provisioned to end users in the form of virtual machines. The Cloud OS manages the full lifecycle of VMs using a virtual machine manager [66, p. 67ff]. This includes deployment, migration, and suspension of VMs via the underlying virtualization technology (see Section 2.1.1). To monitor VM state and resource usage—such as CPU, memory, disk, bandwidth, and power consumption—the Cloud OS employs an information manager. Additionally, a service manager is responsible for handling applications deployed across multiple interconnected VMs.

Given that a datacenter comprises numerous physical machines capable of hosting VMs, a scheduler is required to allocate VMs effectively [66, p. 67ff]. Scheduling occurs in two phases: first, assessing whether a VM can be provisioned with the required resources, and

second, selecting the physical host for deployment. Scheduling decisions can be influenced by specific hardware requirements such as CPU and memory capacity. Efficient VM scheduling is a fundamental challenge in cloud computing and is discussed in more detail in the next section.

In addition to the core VM-related functionalities, the Cloud OS includes other essential components such as a network manager, accounting and auditing systems, authentication and authorization services, storage and image managers, and administrative tools [66, p. 67ff].

4.1.2 Virtual Machine Placement

The allocation of virtual machines to physical hosts in cloud datacenters is an optimization problem. It seeks to balance quality of service and energy efficiency while fulfilling as many customer requests as possible [53, p. 1f]. Virtual machine placement is considered one of the primary challenges in managing virtualized datacenters [67, p. 38].

In this context, virtual machines are typically characterized by four parameters: the number of CPU cores, the CPU capacity per core in MIPS (Million Instructions Per Second), memory, and disk size [53, p. 4].

The VM placement problem is defined by the following conditions: VMs must be assigned to physical hosts, each of which has a finite capacity of resources. The number of concurrently active VMs changes over time, and the resource demands of individual VMs may vary dynamically. Failing to meet resource requirements or service-level guarantees incurs operational costs for the cloud provider [53, p. 3].

Numerous studies have surveyed VM placement strategies aimed at optimizing different objectives [67, 53, 52]. A comprehensive review of these strategies is beyond the scope of this thesis. As previously stated, this thesis focuses primarily on spot instances. The following sections will therefore examine the lifecycle and behavior of spot instances in greater detail.

4.1.3 Spot Instance Life-cycle

Pham et al. (2018) introduced a life-cycle model for spot instances that outlines the key phases and events associated with their use. This model is based on Amazon EC2 Spot Instances [5]. To initiate a spot instance, a user must issue a spot request after selecting a machine type. Additionally, the user must specify a maximum bid price and whether the request should be persistent [33]. A spot request can end in one of three possible outcomes: (1) unfulfilled spot request, (2) fulfilled spot request with interruption, or (3) fulfilled spot request without interruption [5, p. 74].

While waiting for the request to be fulfilled, several conditions can result in an unfulfilled state: the user may cancel the request, the request may reach a deadline and be automatically canceled, or the request may be rejected due to insufficient capacity or a bid price that is too low. If the request is successfully fulfilled, the spot instance is deployed. During its execution, the instance may still be interrupted due to insufficient

capacity or a bid price falling below the market rate, resulting in a fulfilled request with interruption [5, p. 74].

Notably, when an instance is interrupted, it does not stop immediately. Instead, it continues running for a short duration known as the *warning time*. In the experiments conducted by Pham et al. (2018), this warning time was consistently two minutes across all tested regions [5, p. 79]. Alternatively, if the user terminates the instance, the result is a fulfilled request without interruption. A graphical representation of the spot instance life-cycle is shown in Figure 4.1.

Spot Instance Interruption Behavior. Spot instances may be interrupted at any time by the cloud provider. Two primary reasons for interruption exist: (1) insufficient computing capacity—where resources are reallocated to higher-priority On-Demand or Reserved instances—and (2) a bid price lower than the current spot price [5, p. 74].

Pham et al. (2018) conducted experiments to analyze spot instance behavior. Regarding the waiting time between the submission of a spot request and its fulfillment, they found that 80%–90% of requests were fulfilled within 4 seconds. However, if fulfillment did not occur within that time, waiting times typically exceeded 60 seconds. The longest recorded fulfillment time was 3.96 hours. Importantly, no correlation was observed between waiting time and interruption frequency. Once fulfilled, instances ran for a minimum of approximately four minutes before any interruption occurred.

The study also examined whether workload—defined by CPU and memory usage—impacted interruption frequency. No significant difference in interruption rates was found between instances with high and low workloads.

Regarding pricing, between 2.2% and 29.7% of interruptions were attributed to low bid prices. However, no strong correlation between bid price and interruption frequency was identified. Interruptions appeared to be uniformly distributed across different bid values. This suggests that when AWS reclaims capacity due to insufficient resources, the bid price does not influence which spot instances are interrupted [5].

As noted in Section 2.1.3, recent changes to AWS spot instance pricing have resulted in greater price stability, which may further reduce the likelihood of price-based interruptions. For the purposes of this thesis and the corresponding simulation, it is therefore assumed that the bid price has an insignificant impact on interruption probability.

Although the internal algorithm AWS uses to determine which spot instances to interrupt is not publicly disclosed, AWS does publish interruption frequency data for various instance types across regions [68].

Based on the contextual understanding of cloud resource models, the following subsections define the functional and non-functional requirements guiding the system design.

4.2 Functional Requirements

This subsection formally defines the essential capabilities the system must provide to realistically model and manage spot instances in a simulated cloud environment.

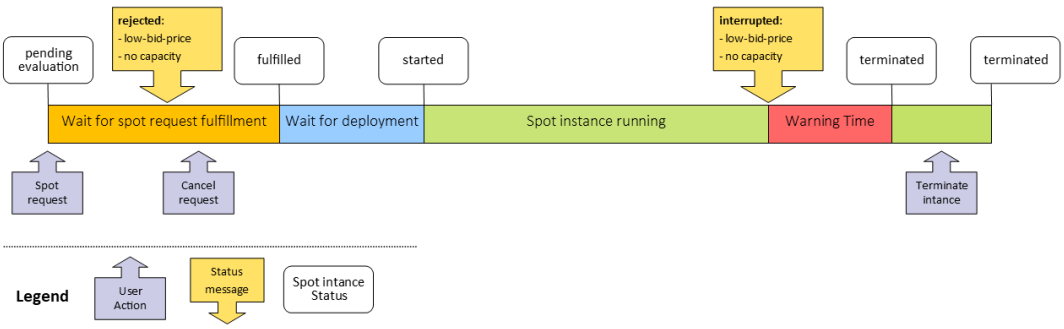

Figure 4.1: *Spot instance life-cycle based on EC2, including events, status messages, and user actions [5]*

- **VM Allocation Behavior:**
  - Spot instances must be interruptible at any time due to on-demand capacity demands.
- **VM Instance Types:**
  - It must be possible to create distinct types of virtual machine instances (e.g., on-demand and spot) with differentiated runtime behaviors.
- **VM Lifecycle Extensions:**
  - If a VM cannot be allocated due to insufficient resources, it should be retained and eligible for resubmission when resources become available.
- **Interruption Handling:**
  - Spot instances must support interruption events that result in either HIBERNATION or TERMINATION.
  - A configurable minimum runtime must be enforced before interruptions are permitted.
- **Resubmission Logic:**
  - Spot instances in HIBERNATION must be periodically resubmitted, depending on resource availability and a user-defined timeout.
  - Unallocated VM requests must be retained for a configurable time before being discarded.
- **Simulation Output and Monitoring:**
  - The framework must be able log execution history for each Spot instance, including active durations and interruption events.

  - Statistics such as the number of interruptions per instance and average interruption time must be available for post-simulation evaluation.

Requirement Rationale

Virtual machines can generally be provisioned using three purchasing models: on-demand, reserved, and spot instances. However, the reviewed simulation frameworks predominantly support only a single VM type—typically representing the on-demand model. The implementation presented in this thesis therefore extends an existing cloud simulation framework to support the distinctive characteristics of spot instances.

Due to fluctuating resource availability of spot instances, the framework must support configurable interruption behavior, including termination and hibernation. In line with common cloud provider guarantees, a minimum runtime must be enforced before interruptions occur.

Furthermore, to reflect the retry logic used in actual cloud environments, unallocated or hibernated instances must be eligible for resubmission based on user-defined constraints. Finally, in order to support robust evaluation, the system must track execution timelines, interruption frequencies, and durations, allowing users to derive meaningful performance metrics post-simulation.

4.3 Non-Functional Requirements

This section outlines the non-functional requirements for implementing spot instance behavior in an existing cloud simulation framework.

- **Modularity:**
  - The chosen cloud simulation framework should provide a modular architecture to allow isolated development of new features.
- **Extensibility:**
  - The framework should support the integration of new functionality without requiring invasive changes to the existing source code.
  - It must be possible to incorporate advanced VM allocation algorithms, such as interruption-aware placement strategies.
- **Maintainability:**
  - The codebase should be logically organized and supported by clear abstractions to simplify future extensions and debugging.
- **Documentation and Transparency:**
  - The framework should be open-source with accessible and well-documented code to ensure comprehensibility and ease of adaptation.
- **Scientific Relevance:**

  - The simulation framework should be widely recognized and used in academic research, supporting reproducibility and comparability with related work. Preference is given to frameworks, that have been cited extensively in peer-reviewed publications.

- **Configurability:**
  - The spot instance model must support user-defined parameters such as minimum runtime, hibernation timeout.

- **Observability:**
  - The solution must provide mechanisms to track and log spot instance state transitions, interruptions, and resubmissions to facilitate evaluation.

Requirement Rationale

To support the modeling of dynamic and interruption-prone behavior, the selected cloud simulation framework needed to be both technically accessible and structurally adaptable. An open-source foundation with clear documentation was essential for understanding internal mechanisms and safely extending functionality. Modularity ensured that new components—such as spot instance behavior and interruption handling—could be developed independently without affecting the stability of core features.

Extensibility was a key requirement, as the integration of advanced scheduling and allocation algorithms demanded flexibility in modifying resource provisioning logic. Given the simplistic nature of default strategies in many frameworks, the ability to introduce and evaluate interruption-aware placement mechanisms was critical. Additionally, maintainability was necessary to ensure that future enhancements, debugging, and experimentation could be carried out with minimal friction. Together, these non-functional requirements formed the foundation for a sustainable and research-oriented implementation approach.

5 Modeling Spot Instances in CloudSim Plus

The objective of the this section was to model and implement an extension for the existing cloud simulation framework, CloudSim Plus, with the primary goal of enabling the simulation of spot instances and their associated dynamic behavior. This section elaborates on the relevant components of the simulation framework and details the newly developed classes and functionality added to support spot instances. The version of CloudSim Plus used for this implementation is `6.2.10`.[1]

CloudSim Plus is an open-source simulation framework developed as an independent fork of CloudSim 3, the most widely used framework in cloud simulation research. Silva Filho et al. (2017) identified several limitations in CloudSim's implementation, which they aimed to address with the development of CloudSim Plus.

One major shortcoming of CloudSim was its poor coding practices and project structure. It lacked proper design patterns, violated established software engineering principles, and contained a high degree of code duplication [69, p. 400]. A code duplication analysis showed that CloudSim Plus reduced duplicated code lines by 30.44% to 2,536 lines, while CloudSim 4 exhibited an increase of 300.96%, resulting in 10,973 duplicated lines [69, p. 403]. These issues negatively impact maintainability, testability, and extensibility.

CloudSim Plus also introduced integration tests to improve test coverage. Integration testing validates the interactions between components and ensures that results are correct and consistent, thus enhancing simulation accuracy [69, p. 403f]. In contrast, CloudSim lacked both sufficient documentation and readily available test results. While both frameworks include Javadoc comments in the source code, CloudSim Plus supplements this with detailed documentation on its website, including FAQs and usage instructions [57]. Additionally, Javadocs must be generated manually for CloudSim, whereas CloudSim Plus offers them pre-generated and accessible online [70].

Another significant advantage of CloudSim Plus is its improved usability. It requires fewer lines of code to create simulation scenarios and provides better performance in execution time when compared to CloudSim 4. Benchmarks show that CloudSim Plus consistently outperforms CloudSim 4 across various virtual machine allocation policies [57].

CloudSim Plus also introduced several exclusive features:

- A trace file reader capable of processing workload data from the Google Cluster Data [71]

[1] The complete source code for the implementation is available at `https://github.com/CGoldi/cloudsimplus6-spot-instance.git`.

- Dynamic creation of virtual machines and cloudlets at runtime, without requiring new brokers
- Support for delayed submission of cloudlets and virtual machines to simulate dynamically arriving tasks
- Event listeners that allow custom actions to be triggered during specific events (e.g., allocation or deallocation of virtual machines)
- Runtime creation of hosts and host failure injection to simulate hardware faults
- Support for continuing simulation in idle periods to await dynamically generated events

Choice of Programming Language. For this project, the choice of programming language was primarily dictated by the selected simulation framework. Since CloudSim Plus is implemented in Java, it was the natural and most compatible choice for the extension development. Using Java ensures seamless integration with the existing codebase, allows reuse of existing components, and simplifies the process of extending the framework's core functionality. The majority of cloud simulation frameworks and tools are developed using the Java programming language. This trend has been confirmed in several comparative studies of simulation tools [13, 40, 72]. One likely reason for this is the widespread use of CloudSim, which itself is implemented in Java and has influenced many derivative tools.

5.1 Core Simulation Components of CloudSim Plus

The following classes form the foundation of the CloudSim Plus simulation engine. These components manage event scheduling and dispatching, and serve as the communication backbone between simulation entities.

CloudSim. The `CloudSim` class is the main entry point of the simulation framework. It is responsible for initializing the simulation, managing the simulation clock, processing scheduled events, and terminating the simulation when all events have been executed or a predefined simulation time limit has been reached. Events are stored in a future events queue, sorted by timestamp. When their scheduled time arrives, events are transferred to the deferred queue for processing and execution [4].

SimEntity. The `SimEntity` interface defines the contract for all simulation entities. These entities can schedule, receive, and handle simulation events, and can communicate with other entities by sending events. All simulation entities in CloudSim Plus must extend the abstract class `CloudSimEntity`, which implements the core logic for event management. Examples of entities include `Datacenter`, `DatacenterBroker`, and `CloudletScheduler`.

SimEvent. The `SimEvent` class defines the structure of an event in the simulation. Each event contains metadata such as the event type, timestamp, source and destination entities, a tag indicating the purpose of the event, and an optional object payload. The object payload varies depending on the event tag and is used to pass additional information between entities.

CloudSimTags. The `CloudSimTags` class defines a set of static constants that serve as unique identifiers for simulation events. These tags determine how each event should be interpreted and processed by the receiving entity. Example tags include `END_OF_SIMULATION` (used to terminate the simulation), `CLOUDLET_SUBMIT` (used to submit a cloudlet to a virtual machine), and `VM_ DESTROY` (used to request the destruction of a virtual machine).

5.2 Key Infrastructure Classes in CloudSim Plus

This section presents the key infrastructure classes of the CloudSim Plus framework, which are essential to understanding the simulation model and the basis for the implementation work discussed in later sections [70, 69].

DatacenterSimple. This class represents a virtualized cloud datacenter and provides the foundational components for modeling cloud infrastructure. A `DatacenterSimple` instance contains a set of `Host` objects that define the physical resources available. Each datacenter is also associated with a VM allocation policy, which determines how virtual machines are placed across hosts.

VmAllocationPolicyAbstract. This abstract class defines the logic for allocating virtual machines to hosts and optionally migrating them during simulation. It is also responsible for managing vertical scaling of VM resources. Developers can extend this class and override the `defaultFindHostForVm()` method to implement custom allocation strategies. CloudSim Plus already includes several basic allocation policies, including `FirstFit`, `BestFit`, `Random`, and `RoundRobin`.

HostSimple. This class models a physical machine within a datacenter. Each host has configurable resources including RAM, bandwidth, storage, and processing elements (PEs), which represent CPU cores. Hosts can manage multiple VMs and provide different scheduling options. CloudSim Plus supports both time-shared and space-shared scheduling policies for allocating PEs among VMs.

DatacenterBrokerAbstract. The datacenter broker simulates the behavior of software agents acting on behalf of cloud users. It manages VM lifecycle operations such as creation, submission, and destruction, and handles cloudlet scheduling. The broker can be customized by extending `DatacenterBrokerAbstract` to implement different policies for allocating cloudlets to VMs.

VmSimple. This class represents a virtual machine that can be deployed on a `Host`. Similar to physical hosts, each VM has specific resource requirements such as RAM, storage, bandwidth, and number of PEs. VMs execute cloudlets and utilize a `CloudletScheduler` to manage task execution. Scheduling options include time-shared and space-shared models.

Cloudlet. `Cloudlet` instances represent the computational tasks or applications running inside VMs. The workload of a cloudlet is defined by the number of required PEs and the total number of instructions (in MIPS) it must execute. For other resources such as RAM and bandwidth, different utilization models can be applied. The default is `UtilizationModelFull`, where the cloudlet uses the full capacity of all assigned resources throughout its execution.

EventListener. Event listeners allow simulation components to react to specific changes in system state. Each listener monitors a particular type of event and can be attached to various entities, including datacenters, brokers, hosts, VMs, cloudlets, schedulers, or the main simulation instance. The framework provides several predefined listener types, such as:

- `onClockTickListeners` – triggered on every simulation time update.
- `onFinishListeners` – invoked when a cloudlet finishes execution.
- `onHostAllocationListeners` – called when a VM is allocated to a host.

These listeners provide a flexible mechanism for extending simulation functionality by executing custom actions in response to runtime events.

5.3 Modeling Objectives

This section introduces the modeling objectives of the spot instance extension and illustrates how the design abstracts spot instance behavior within the CloudSim Plus simulation framework. The aim is to conceptually map the new functionality onto the architecture of the existing simulation environment and to explain how the new components interact with core elements such as the virtual machine lifecycle, allocation policies, and simulation control flow.

Figure 5.1 provides a high-level component view of the integration. The yellow box highlights the original CloudSim Plus modules, while the orange box labeled “Spot Instance Modeling” encapsulates the new components introduced by the extension. These components implement core functionality required to simulate dynamic spot instance behavior, including lifecycle transitions, interruption logic, and resubmission mechanisms. The diagram also reflects how spot-specific components interface with host scheduling and VM allocation policies during simulation execution.

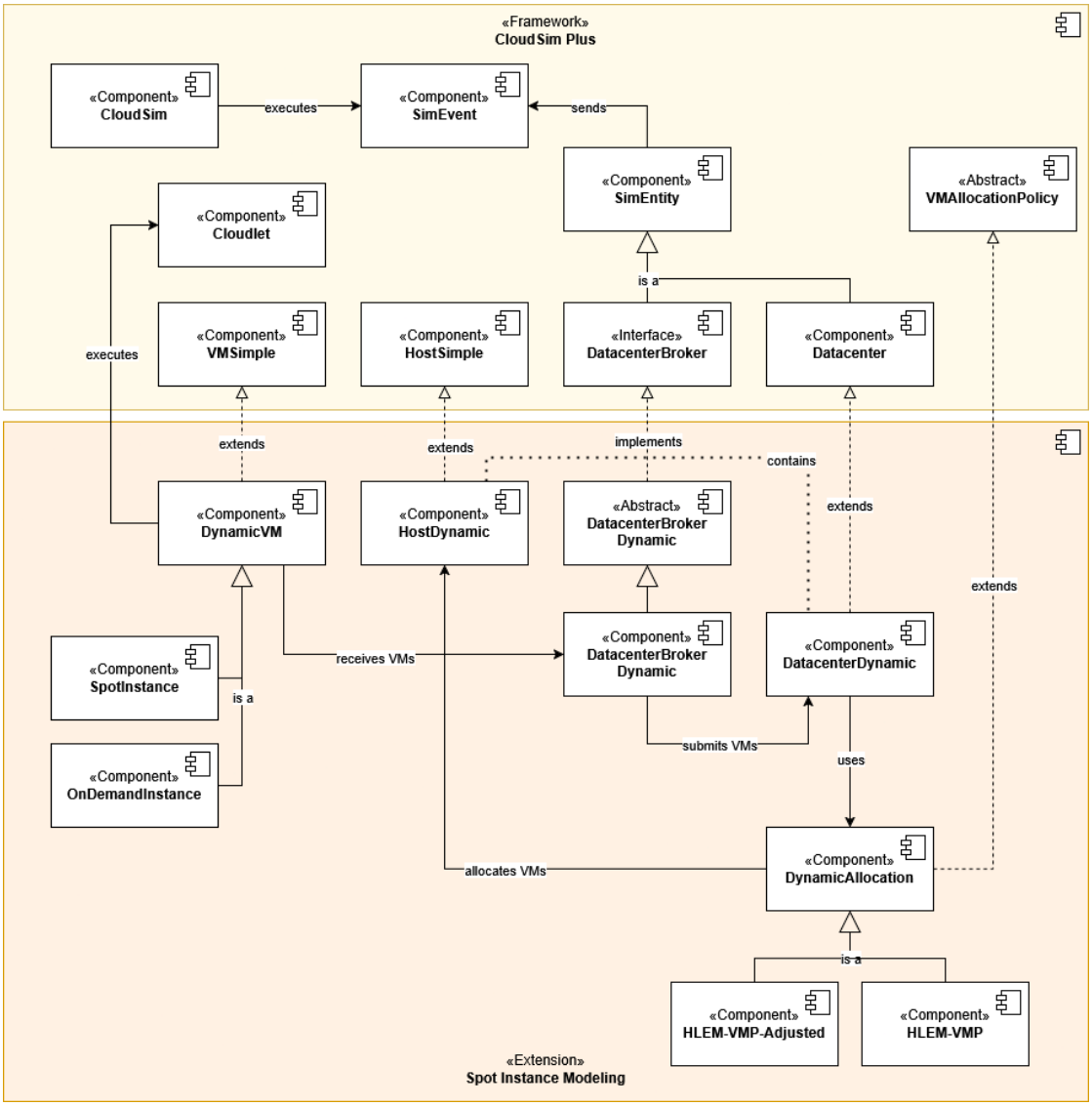

Figure 5.1: *Component diagram of the Spot Instance Modeling extension*

To complement the structural model, two activity diagrams depict runtime behavior within the simulation. The first (Figure 5.2) illustrates the allocation process of on-demand VMs, while the second (Figure 5.3) captures the state transitions and lifecycle management of spot instances.

CloudSim Plus, by default, supports dynamic VM arrivals but immediately discards requests that cannot be fulfilled due to insufficient resources. The spot instance extension overrides this behavior by modeling persistent VM requests and enriching the lifecycle with configurable interruption events and recovery mechanisms. These enhancements allow for a better representation of the uncertainty and volatility associated with spot pricing models in commercial cloud markets.

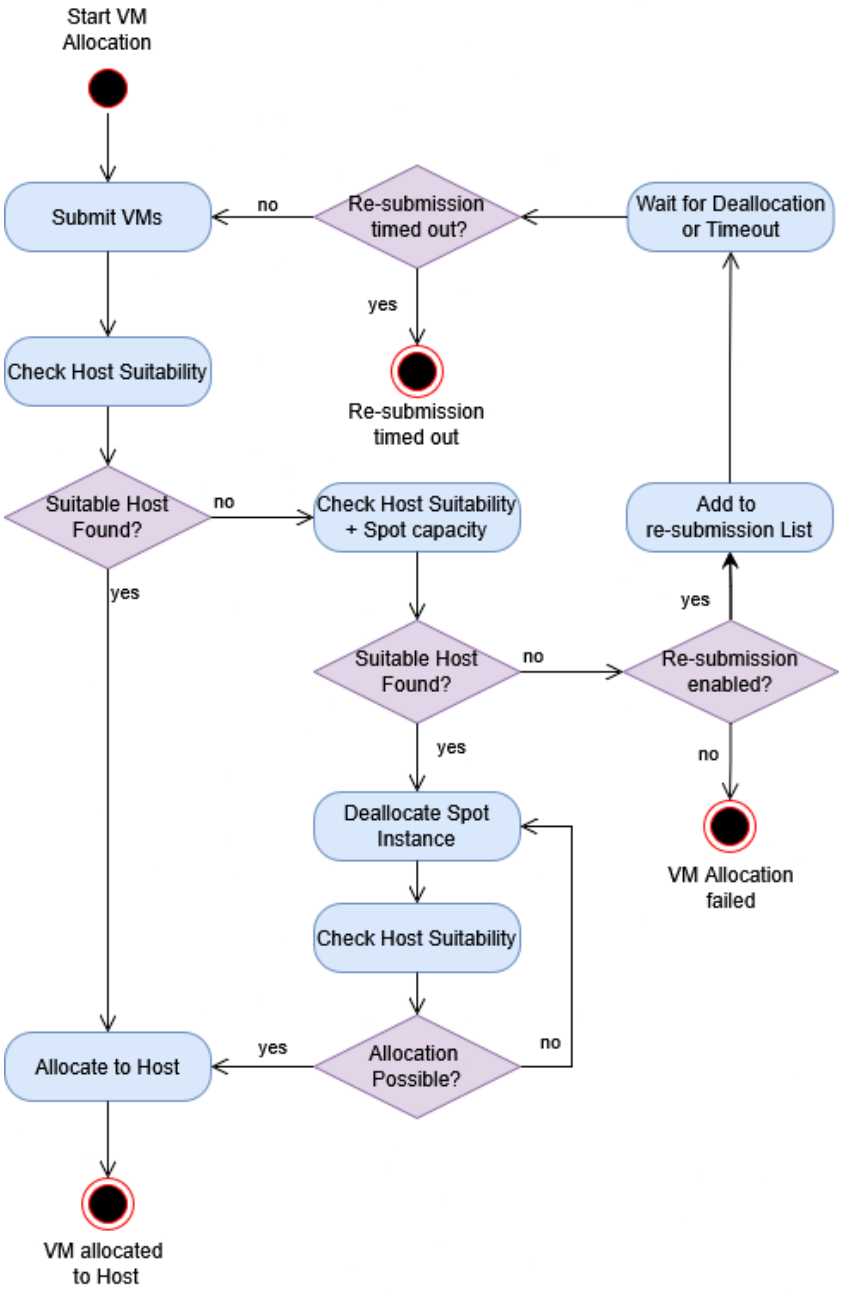

Figure 5.2: *Activity diagram: VM allocation*

Given the thesis's focus on spot instance behavior, several new features were developed to simulate their lifecycle more realistically. One critical addition is the support for dynamic termination and hibernation. When an on-demand instance request cannot be fulfilled due to capacity constraints, the system attempts to free up resources by

interrupting spot instances. The selection of which host to target for interruption depends on the active VM allocation policy.

Each spot instance can be configured with an interruption behavior: either *termination* or *hibernation*. In the case of termination, the instance is immediately destroyed. If hibernation is selected, the instance is temporarily removed from the host, and its associated cloudlets are paused. Once sufficient capacity becomes available again, the spot instance is reallocated to a host, and its cloudlets resume execution.

The implementation also includes tracking mechanisms for spot instance execution history. Each period of activity is recorded, allowing for the calculation of average interruption times and other metrics. To support realistic and configurable simulation behavior, the following time-based parameters were introduced:

- **Waiting Time:** Defines how long a persistent request remains active before being discarded.

- **Hibernation Time:** Specifies the maximum duration a spot instance may remain in hibernation before being terminated.

- **Minimum Running Time:** Ensures that spot instances cannot be interrupted due to capacity constraints until they have been running for a specified minimum duration.

- **Warning Time:** Defines the delay between an interruption signal and the actual termination, simulating a grace period.

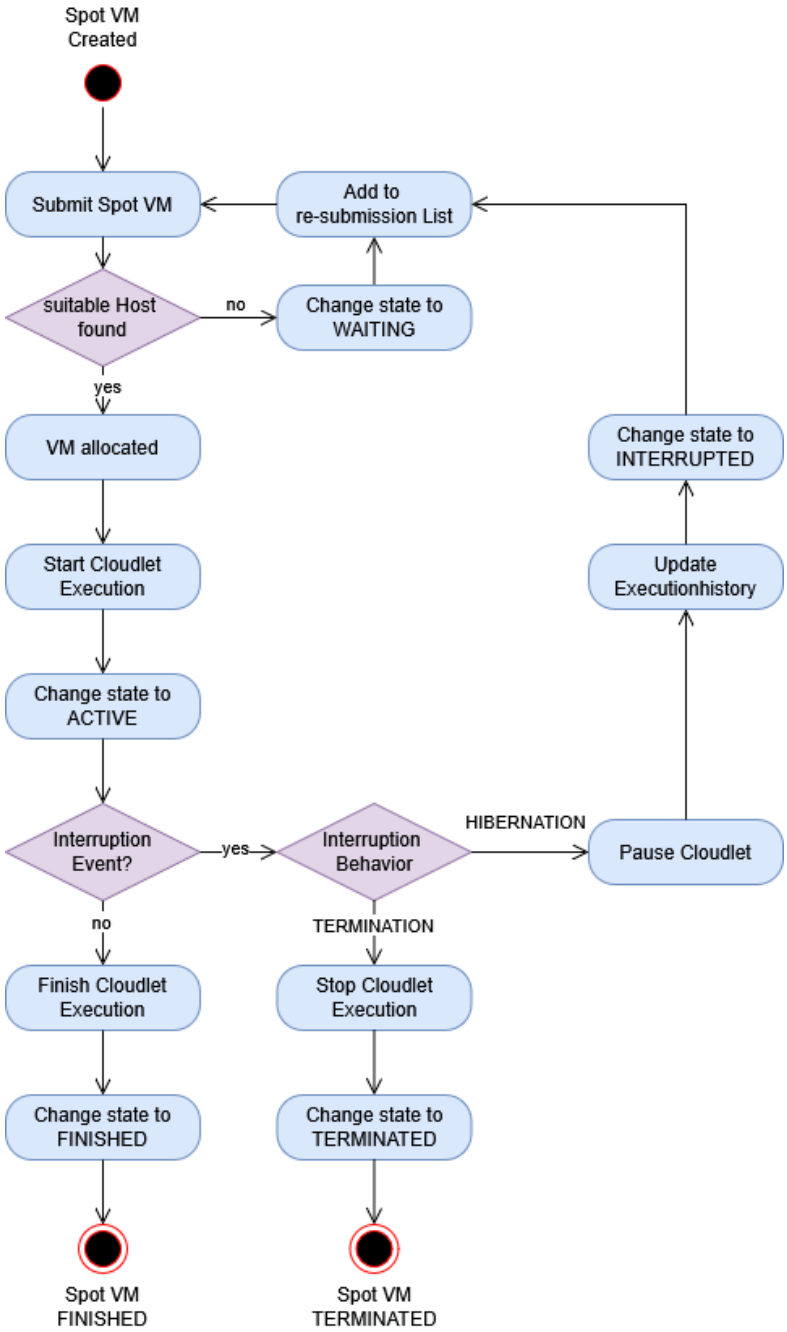

Figure 5.3: *Spot Instance Lifecycle*

5.4 Spot Behavior Extension for CloudSim Plus

To support the simulation of spot instances, several key features were added to extend the capabilities of CloudSim Plus. The primary enhancement is the *differentiation of virtual machine types*, specifically the addition of on-demand and spot instances. Spot instances include advanced behaviors such as *automatic termination or hibernation* when resources are needed for on-demand instances. Hibernated spot instances can be *automatically resumed* once sufficient resources become available. The interruption behavior (termination or hibernation) and resubmission time window can be configured individually for each spot instance.

Additionally, *persistent allocation requests* were introduced. In the standard CloudSim Plus framework, if a virtual machine cannot be allocated immediately, the request is discarded. In contrast, the extended implementation allows users to define a *waiting time* during which the request remains active. If sufficient resources become available within this window, the virtual machine will be instantiated.

5.5 Newly Introduced Classes

In this section, the classes that got newly introduced to implement the new features added as an extension to Cloudsim Plus. First, the purpose of the class will be explained and a relation to the existing classes of CloudSim Plus will be made. Furthermore, the most important instance variables as well as the methods of the classes will discussed in detail.

DataCenterBrokerDynamic. The datacenter broker is responsible for submitting and managing the virtual machines and cloudlets. The new datacenterbroker provides additional functionality, that allows to dynamically resubmit virtual machines, that have been interrupted or couldn't be allocated immediately. It extends the class DataCenterBrokerAbstract. The following instance variables and methods have been implemented in this class:

- **resubmittingList:** VM's that aren't able to be created or Spot instances that get interrupted are added to this resubmitting List

Methods:

- resubmitVms: Resets the last selected datacenter and the allocation data from the virtual machine instance to enable a new allocation. The VMs from the resubmitting List and will be resubmitted by the Broker, which restarts the allocation of the intances if possible
- resetBrokerAllocation: Resets the last selected Datacenter, so new VM's can be created by the same broker after deallocation of other VMs
- requestWaitingCloudlet: This method was added to access the private method requestDatacentersToCreateWaitingCloudlets from DatacenterBrokerAbstract.java, the access was needed to enable the creation of Instances after a creation failure

DynamicAllocation. The DynamicAllocation class contains the main logic for allocating virtual machine and interrupting to free capacity spot instances. DynamicAllocation extends the abstract class VMAllocationPolicyAbstract. The following methods have been implemented in this class:

- allocateHostForVM: Tries to find a suitable host for a virtual machine instance and allocates the instance to it if it is found. If no suitable host is found, it will be checked if any hosts have enough free capacity if spot instances get deallocated. When after this step still no suitable host was found, the virtual machine will be added to the resubmitting List. Overrides the method allocateHostForVm from VMAllocationPolicyAbstract.java.
- spotAllocation: Tries to find a host that has enough capacity for launching a new Virtual Machine instances by deallocation Spot instances. If a suitable host is found the capacity gets freed by terminating or hibernating these spot instances.
- freeCapacity: If a suitable host was found, spot instances will be removed from the host until enough capacity is freed to allocate the new virtual machine
- checkSpotCapacityUsage: Checks the total amount of resources that are used by Spot instances on a specific Host, this is done to avoid terminating instances that wouldn't provide enough space for the Virtual Machine instance that has to be allocated
- terminationBehavior: Depending on the interruption behavior of the Spot instance, it will either be interrupted or terminated. If the behavior is set to HIBERNATE, the Spot instance will be INTERRUPTED and all running Cloudlets will be paused. If the behavior is set to TERMINATE, the Spot instance will be TERMINATED and the instance including the cloudlets will be destroyed.
- spotAllocationSpecificHost: Enables the allocation of virtual machines to a specific Host, instead of finding a suitable host through the allocation policy. Destroys spot instances to free capacity if necessary. Main purpose of this method is the Google Trace Data example, which you can see in section 7.3.

DynamicVm. This class is an abstract class, that provides additional functionality to the existing virtual machine instances, it extends the SimpleVm class. The DynamicVm class is the base class for the two virtual machine types that have been implemented. The following instance variables and methods have been implemented in this class:

- **persistentRequest:** Determines if a request will be persistent or not. If a request is persistent, additional requests will be made if the request fails or if the instance gets interrupted. By default creation requests are not persistent.
- **waitingTime:** This variable defines the time until a request will be canceled if it is not full-filled. By default requests get cancelled immediately if the creation fails.

- **initialRequestTime:** Saves the time of the first time the instance sends a request to get allocated.
- **pausedCloudlets:** A list of paused cloudlets that are assigned to this virtual machine instance
- **failedCloudlets:** A list of failed cloudlets that are assigned to this virtual machine instance
- **state:** Within the class DynamicVm is an enumeration called **State**, that determines the current state of the virtual machine. There are six possible states that can be achieved:
  - *WAITING:* A virtual machine instance that is already created, but has not been assigned to a host yet, will be put in the waiting state. This is mostly used for instances that couldn't be allocated immediately but have persistent requests activated.
  - *ACTIVE:* Virtual machine instances are active if they are currently running on a host
  - *FAILED:* If a virtual machine instance is not able to be created, it will be put in the failed state. Reasons for this can be that no suitable host is available.
  - *INTERRUPTED:* Spot instances can be shut down anytime to free resources for on demand instances. The behavior of the Spot instances can be determined by the user. If the interruption behavior is set to hibernate, the spot instance will be paused and reach the interrupted state. Interrupted spot instances will be automatically resumed at a later point if enough capacity is available again.
  - *TERMINATED:* If the user set termination as interruption behavior, the spot instance will be shut down and reach the terminated state when capacity is needed for other virtual machine instances
  - *FINISHED:* When an instance has executed all scheduled workloads it will be shut down and reach the finished state

Methods:

- allocationBehavior: This method is an allocation listener for Dynamic Virtual Machine instances. When an virtual machine instance gets allocated to a host the state of the instance will change to ACTIVE and all paused Cloudlets will be resumed and removed from the pausedCloudlet list. Will be activated if an allocation event for this instance occurs during the simulation.
- deallocatingBehavior: An deallocation listener for Dynamic Virtual Machine instances. When an instance gets deallocated and it didn't get INTERRUPTED or TERMINATED the state gets set to FINISHED. Will be activated if an deallocation event for this instance occurs during the simulation.

- creationFailedBehavior: Creation Failure listener for a virtual machine instance. If the allocation of the instance fails, the state gets set to WAITING if it has an assigned waitingTime bigger then zero, otherwise it gets set to FAILURE

OnDemandInstance. The OnDemandInstance class models on demand virtual machine instances, it is one of the two classes that extend the DynamicVM class. On demand instances don't have any additional functionality so far, but are necessary to differentiate which instances are able to be interrupted to free resources.

SpotInstance. This class aims to model spot instances virtual machine instances, it extends the DynamicVM class. The following instance variables and methods have been implemented in this class:

- **interruptionBehavior**: For the variable interruptionBehavior an enumeration class (**InterruptionBehavior**) was introduced to determine how the instance behaves in case of an interruption.
  - *TERMINATE:* When terminate is chosen as the interruption behavior, the spot instance will be shut down if the resources are for other instances.
  - *HIBERNATE:* When terminate is chosen as the interruption behavior, the spot instance will be paused if the resources are needed for other instances and can be resumed later.

  The default interruption behavior of a spot instance is terminate.
- **minimumRunningTime:** This variable defines the minimum time an virtual machine instance needs to be active before it can get interrupted, the default value is 0
- **warningTime:** The warning time specifies how long an DynamicVm instance remains active after it gets deallocated, the default value is 0
- **averageInterruptionTime:** Displays the average time, the spot instance was interrupted, is calculated with the Executionhistory values
- **executionHistory** List of Executionhistory entries, shows the times the spot instance was active
- **hibernationTimeLimit:** Time limit for the hibernation, defines how long a spot instance remains hibernated before it gets if it gets interrupted
- **maxBidPrice:** Defines the maximum bid price of the spot instance, this variable is added for completeness but interruption for too low bid prices isn't part of the implementation, see section 4.1.3 for more detail information

Methods:

- calculateAverageInterruptionTime: Calculates the average interruption time based on the execution history
- updateExecutionHistory: Deallocation Listener that updates the Execution history of a spot Instance each time they get deallocated.

ExecutionHistory. The ExecutionHistory class is responsible for tracking each individual execution of a spot instance, which includes the host, start time and stop time. If a spot instance gets interrupted and resubmitted a new execution history entry is made. The execution history is essential to calculate the interruption time of the spot instances. The follwing insance variables have been implemented in this class:

- **startTime:** The start time of the spot instance execution
- **stopTime:** The stop time of the spot instance execution
- **host:** The host on which the spot instance was running during the execution

DynamicVmTableBuilder, SpotVmTableBuilder, ExecutionTableBuilder. Custom table builders extending `TableBuilderAbstract` to summarize simulation results.

- **DynamicVmTableBuilder:** Lists info for all dynamic VMs, including timing, resource, and state details.
- **SpotVmTableBuilder:** Specializes in spot VMs; includes average interruption time.
- **ExecutionTableBuilder:** Displays execution intervals of spot instances; includes JSON export functionality.

5.6 Modified Classes

One of the primary objectives of this implementation was to extend CloudSim Plus without altering its existing core classes. Maintaining compatibility with the original framework ensures that the extension remains modular, maintainable, and easy to integrate or update in the future.

TableBuilderAbstract. This class serves as the base for all table builder classes used to present simulation results in a human-readable table format in the console. To enhance the usability of simulation output, especially for large-scale experiments or post-simulation analysis, a method was added to enable saving the table data to a `.csv` file. This allows users to process results externally, for instance in spreadsheet software or data analysis tools.

Additionally, for the implementation of the *Google TraceReader* example (discussed in section 7.3), minor adjustments were required in a few existing classes. However, these

modifications do not affect the simulation core of CloudSim Plus and are isolated to application-specific functionality.

5.7 Reflections on Design Choices

This section provides a critical evaluation of the design decisions made during the development of the simulation extension. It addresses both the necessity of the implementation and the rationale behind selecting specific technologies and tools.

Developing a comprehensive cloud simulation framework from scratch was deemed infeasible within the scope of this thesis due to the significant time and resource requirements. Consequently, the alternative approach was to extend an existing simulation framework that already supports core infrastructure modeling and offers extensibility. As discussed in the previous sections of this chapter, CloudSim Plus was selected for this purpose. It is an actively maintained open-source framework and a well-structured fork of CloudSim that resolves many of its predecessor's limitations, such as high code duplication, lack of integration testing, and poor adherence to software engineering principles [69]. CloudSim Plus also introduces advanced features, including support for dynamic workloads, runtime entity creation, trace file reading, and flexible event listeners, making it a suitable foundation for extending cloud simulation capabilities with spot instance support.

While the rationale for choosing Java as the programming language was previously discussed, it is worth reflecting on its strengths and limitations. Java's strong support for object-oriented programming, rich ecosystem of libraries, and active community make it a suitable choice for building extensible and maintainable simulation tools. Furthermore, Java's mature tooling ecosystem facilitates debugging, testing, and integration, which are essential for research-focused software development.

Nevertheless, Java is not without its drawbacks. Compared to languages like C++, Java can incur higher memory consumption and longer execution times. However, these drawbacks were considered acceptable in the context of this thesis, given the priority placed on code readability, maintainability, and ease of integration with the selected framework.

In summary, the design choices were guided by practical constraints and the need for a robust, extensible simulation platform. CloudSim Plus provided a solid foundation for implementing and evaluating new functionality such as spot instance behavior, while Java ensured seamless integration and maintainability of the resulting extension.

6 Allocation of Virtual Machines in Dynamic Spot Markets

The efficient placement of virtual machines is a central challenge in cloud computing, particularly when simulating dynamic environments such as those involving spot instances. VM allocation algorithms play a critical role in determining which physical host is assigned to a given VM request, directly impacting resource utilization, energy efficiency, and service continuity.

This chapter addresses the second research question posed in Chapter 3, which investigates how VM placement strategies can be adapted to reduce spot instance interruptions and improve overall resource efficiency in dynamic cloud environments.

The implemented approach builds on and extends the HLEM-VMP algorithm, which incorporates resource load balancing and energy-aware decisions into the placement process. The chapter is structured as follows: Section 6.1 introduces the conceptual foundation of the HLEM-VMP algorithm. Section 6.2 discusses the implementation of the algorithm within the CloudSim Plus framework. Finally, Section 6.3 describes enhancements introduced to reduce spot instance interruptions and improve fairness across physical hosts.

6.1 HLEM-VMP: A Heuristic Allocation Algorithm

To support the placement of virtual machines in dynamic cloud environments a VM allocation algorithm was implemented. Care was taken to extend the CloudSim Plus framework without modifying its core classes. This design ensures compatibility with future versions and maintains the modularity and extendability of the original framework. For instance, it remains possible to introduce additional allocation algorithms for both cloudlets and virtual machines.

In selecting a suitable allocation algorithm, the dynamic nature of cloud workloads, particularly when simulating spot instances, was a key consideration. Traditional metaheuristic algorithms, such as Genetic Algorithms or Swarm Intelligence, are often employed to solve static bin packing or resource allocation problems. However, in this thesis, the workloads are time-dependent: virtual machines are dynamically created and terminated, making this a *temporal bin packing problem*.

Applying traditional GA-based approaches in this context would significantly increase complexity, as the algorithm must not only consider multi-dimensional resource constraints (CPU, RAM, etc.) but also the exact timing and duration of VM placement. The evolving simulation environment—with spot instance interruptions, resubmissions, and changing

host availability—further complicates the use of such heuristics.

Instead, a heuristic load- and energy-aware VM placement algorithm was implemented to better address the requirements of a dynamic scheduling environment. HLEM-VMP aims to balance the load across physical hosts and reduce unnecessary energy consumption, while accounting for constraints introduced by spot instance behavior. This enables faster decision-making and greater flexibility during simulation execution.

The HLEM-VMP algorithm consists of three phases: host filtering, host load evaluation, and host selection [63, p. 1966]. In the host filtering phase, only hosts with sufficient free resources to meet the VM's requirements are selected. All resource types—CPU, memory, bandwidth, and storage—are considered. Additionally, unlike the original algorithm, this implementation checks the potential capacity of hosts if active spot instances were to be deallocated. This extension is necessary to accommodate the specific requirements of spot instance management.

To avoid placing VMs with similar workloads on the same host, the algorithm introduces the RsDiff value (Equation 6.1). This is calculated by subtracting the current CPU utilization of the physical host $U_i^1(t)$ from the CPU request of the VM $R_j^1(t)$. Only hosts for which RsDiff exceeds a predefined threshold Thr_{cpu} are considered for VM placement.

$$\text{RsDiff} = R_j^1(t) - U_i^1(t) * Rc \tag{6.1}$$

$$\text{RsDiff} > \text{Thr}_{\text{cpu}} \tag{6.2}$$

The second phase, host load evaluation, involves computing weights for each resource to assess host suitability. First, the standardized available capacity $C_i^d(t)$ of each resource type d on each host i is normalized. Next, the proportion of each resource available on host i relative to all hosts is computed.

$$d_i(t) = \frac{C_i^d(t) - \min\{C_1^d(t), \ldots, C_n^d(t)\}}{\max\{C_1^d(t), \ldots, C_n^d(t)\} - \min\{C_1^d(t), \ldots, C_n^d(t)\}} \tag{6.3}$$

$$p_{id} = \frac{C_i^d(t)}{\sum_{i=1}^{n} C_i^d(t)}, \quad (i = 1, \ldots, n;\ d = 1, \ldots, D) \tag{6.4}$$

The entropy e^d for each resource is then calculated based on p_{id}. Entropy quantifies the uncertainty or distribution imbalance of the resource:

$$e^d = -k \sum_{i=1}^{n} p_{id} \ln(p_{id}) \tag{6.5}$$

The constant k is defined as:

$$k = \frac{1}{\ln(n)} \tag{6.6}$$

Using entropy, the variation factor g^d for each resource is computed. This measures the degree to which the resource influences the overall host evaluation. Weights for each resource are then derived by normalizing the variation factors.

$$g^d = 1 - e^d \tag{6.7}$$

$$w^d = \frac{g^d}{\sum_{d=1}^{D} g^d} \tag{6.8}$$

Finally, the overall host score $\mathrm{HS}_i(t)$ is computed by summing the product of weights and the normalized capacities for each resource:

$$\mathrm{HS}_i(t) = \sum_{d=1}^{D} w^d \cdot C_i^d(t) \tag{6.9}$$

The third and final phase is host selection. Hosts are sorted based on their computed scores, and starting from the host with the best score, each is evaluated for power consumption. If adding the VM does not exceed a predefined energy threshold, the VM is placed on that host [63, p. 1967]. In this implementation, energy consumption checks were omitted for simplicity, and the VM is allocated to the host with the highest score.

The corresponding pseudocode implementation of the HLEM-VMP algorithm is provided in Algorithm 1.

Algorithm 1 HLEM-VMP VM Placement Strategy

Require: Physical Hosts `PHs`, Virtual Machines `VMs`
Ensure: Placement of VMs on PMs

Initialization:
  `PHCandidateList` $\leftarrow \emptyset$
  $Thr_{cpu} \leftarrow 0$, $R_c \leftarrow 0.95$
End Initialization
for each V_j in `VMs` **do**
  `PHCandidateList` $\leftarrow$ `FilterPH(PHs,` V_j`)`
  `PHCandidateListClrSpot` $\leftarrow$ `FilterPHWithSpotClr(PHs,` V_j`)`
  if length(`PHCandidateList`) > 0 **then**
    `HSlist` $\leftarrow$ `CalPHScore(PHCandidateList)`
    Sort `PHCandidateList` by `HSlist(i)` ascending
  else
    `HSlist` $\leftarrow$ `CalPHScore(PHCandidateListClrSpot)`
    Sort `PHCandidateListClrSpot` by `HSlist(i)` ascending
  end if
end for

Implementation specifics and performance evaluations of the HLEM-VMP algorithm are detailed in Sections 6.2 and 7.5.

6.2 Implementation of the HLEM-VMP algorithm

This section introduces the HLEM-VMP (Heuristic Load and Energy-aware VM Placement) algorithm, which was implemented as a custom VM allocation policy in CloudSim Plus. The purpose of this algorithm is to optimize the placement of virtual machines with respect to load balancing and energy efficiency. The placement logic was implemented in the `DynamicAllocationHLEM` class, which extends the functionality of the `DynamicAllocation` class introduced in the previous subsection.

- **resourceCarryingFactor:** The R_c value used in the RsDiff calculation, as shown in Equation 6.1. The default value is 0.95.
- **threshold:** The threshold used for host filtering, as defined in Equation 6.2. The default value is 0.

Methods:

- defaultFindHostForVm: This method contains the main loop of the algorithm. It iterates over all hosts, checks their suitability, and computes the RsDiff value for suitable hosts. For the filtered hosts, the `hostEvaluation` method is subsequently called.
- hostEvaluation: First, the method `calculateCapacityVariables` is called to perform the initial two steps of the algorithm. Then, the entropy e^d, the factor of variation g^d, the weights for each resource w^d, and finally the host scores $\mathrm{HS}_i(t)$ based on the normalized capacities are computed.
- calculateCapacityVariables: This method calculates the standardized available capacity of each host $C_i^d(t)$, as defined in Equation 6.3, as well as the proportional capacity p_{id}, defined in Equation 6.4.

6.3 Enhancing HLEM-VMP for Spot Instance Reliability

While HLEM-VMP provides a heuristic-based VM placement strategy, further refinements were introduced to better align with the simulation goals—specifically, minimizing spot instance interruptions and promoting fairer host utilization. This section describes the adjustments made and the rationale behind them.

To reduce the frequency of spot instance interruptions, the placement strategy aims to distribute spot instances more evenly across physical hosts. This is achieved by calculating the *Spot Load* ($\mathrm{SL}_i(t)$), which reflects the proportional resource usage of spot instances on a given host. The Spot Load is computed as the weighted sum over all resource types, where the weights are determined as described in Equation 6.8:

$$\mathrm{SL}_i(t) = \sum_{d=1}^{D} w^{(d)} \cdot \frac{C_i^{\mathrm{spot},d}(t)}{C_i^{\mathrm{total},d}(t)} \tag{6.10}$$

This Spot Load value is scaled by a tunable parameter α, which defines its influence on host selection. The result is used to adjust the original Host Score ($\mathrm{HS}_i(t)$), yielding an *Adjusted Host Score* ($\mathrm{AHS}_i(t)$) as follows:

$$\mathrm{AHS}_i(t) = \mathrm{HS}_i(t) \cdot (1 + \alpha \cdot \mathrm{SL}_i(t)) \tag{6.11}$$

To implement this logic, the original `DynamicAllocationHLEM` class was duplicated and modified, resulting in the new `DynamicAllocationHLEMAdjusted` class. Key methods were updated as described below:

Adjusted Methods:

- calculateCapacityVariables: After computing the proportional resource usage, the spot-specific capacity usage is calculated for each resource dimension (e.g., CPU, RAM, Bandwidth, Storage).
- hostEvaluation: Before computing the original host scores $\mathrm{HS}_i(t)$, the Spot Load $\mathrm{SL}_i(t)$ is determined using the resource usage values and weights $w^{(d)}$ from Equation 6.10. Once the base host scores are computed, they are adjusted according to Equation 6.11 by applying the Spot Load penalty factor α.

7 Evaluation and Analysis

This section evaluates the developed extension for CloudSim Plus by defining relevant evaluation criteria and testing the implementation in different scenarios. The evaluation focuses on demonstrating the functionality of the extended features, especially the behavior of Spot Instances. To this end, two custom test cases were created. Additionally, real-world data from the Google Cluster Trace is used to assess the implementation under realistic workloads. The goal is to investigate how the simulation environment responds to dynamic instance behavior and cloud market conditions.

7.1 Simulation Setup and Execution

This section describes the steps required to implement and execute a minimal example of the developed extension. In this example, a single datacenter with one host is created to demonstrate the lifecycle of both on-demand and spot instances.

To begin, clone the Git repository containing the extension [1]. A set of ready-to-use test cases is available and discussed in Section 7.2.

Initialize Simulation Components. The following variables are required to initialize the simulation:

```
1 private final CloudSim simulation;
2 private final DatacenterBrokerDynamic broker0;
3 private final List<DatacenterBroker> brokerList = new ArrayList<>();
```

Listing 7.1: *Simulation and Broker Declarations*

First, create the simulation instance using the `CloudSim` class. A minimum time between events and a simulation termination time are set.

```
1 simulation = new CloudSim(0.5);
2 simulation.terminateAt(70);
```

Listing 7.2: *Simulation Setup*

Create Host and Datacenter. A list containing a single host is created. The host is initialized using the `HostDynamic` class with a specified amount of RAM, bandwidth, storage, and processing elements (PEs):

[1] The complete source code for the implementation is available at `https://github.com/CGoldi/cloudsimplus6-spot-instance.git`.

```
final List<Pe> peList = List.of(new PeSimple(1000), new PeSimple(1000));
Host host = new HostDynamic(2048, 10000, 1000000, peList);
```

Listing 7.3: *Create Host*

The datacenter is instantiated using the `DatacenterSimple` class. The host list and a custom allocation policy, `DynamicAllocationHLEM` (based on HELM-VPM), are passed as parameters.

```
final DynamicAllocationHLEM allocationPolicy = new DynamicAllocationHLEM
    ();
Datacenter datacenter0 = new DatacenterSimple(simulation, List.of(host),
    allocationPolicy);
datacenter0.setSchedulingInterval(1);
```

Listing 7.4: *Create Datacenter*

Create Broker. The simulation broker, based on the `DatacenterBrokerDynamic` class, is used to submit VMs and cloudlets to the datacenter.

```
broker0 = new DatacenterBrokerDynamic(simulation);
brokerList.add(broker0);
broker0.setShutdownWhenIdle(false);
broker0.setVmDestructionDelay(1);
```

Listing 7.5: *Create Broker*

Define Virtual Machines. Spot instances are created using the `SpotInstance` class. Configuration parameters such as RAM, bandwidth, storage size, and interruption behavior (e.g., `HIBERNATE`) are set.

```
final SpotInstance spotvm = new SpotInstance(1000, 2, true);
spotvm.setRam(512);
spotvm.setBw(1000);
spotvm.setSize(10000);
spotvm.setInterruptionBehavior(SpotInstance.InterruptionBehavior.
    HIBERNATE);
spotvm.addOnHostDeallocationListener(this::onHostDeallocationListener);
```

Listing 7.6: *Create Spot VM*

On-demand instances use the `OnDemandInstance` class and may be delayed using `setSubmissionDelay`:

```
final OnDemandInstance ondemandvm = new OnDemandInstance(1000, 2, true);
ondemandvm.setRam(512);
ondemandvm.setBw(1000);
ondemandvm.setSize(10000);
ondemandvm.setSubmissionDelay(10);
ondemandvm.addOnHostDeallocationListener(this::onHostDeallocationListener
    );
```

Listing 7.7: *Create On-Demand VM*

Create Cloudlets. Each VM must be associated with a cloudlet that determines its execution workload during the simulation:

```
Cloudlet cloudletSpot = new CloudletSimple(1, 20000, 1);
cloudletSpot.setFileSize(300);
cloudletSpot.setOutputSize(300);
cloudletSpot.setUtilizationModel(new UtilizationModelFull());
cloudletSpot.setVm(spotvm);
```

Listing 7.8: *Create Cloudlet*

Submit VMs and Cloudlets. VMs and cloudlets are submitted to the broker for scheduling and execution:

```
broker0.submitVm(spotvm);
broker0.submitCloudlet(cloudletSpot);
broker0.submitVm(ondemandvm);
broker0.submitCloudlet(cloudletOnDemand);
```

Listing 7.9: *Submit VMs and Cloudlets*

Simulation Loop and Execution. The simulation must update the processing of all active VMs on each clock tick. This is done by registering an event listener:

```
private void updateProcessingforVms(EventInfo eventInfo) {
  for (DatacenterBroker broker : brokerList) {
    for (Vm vm : broker.getVmExecList()) {
      vm.updateProcessing(simulation.clock(), vm.getHost().getVmScheduler
    ().getAllocatedMips(vm));
    }
  }
}
```

Listing 7.10: *Update VM Processing*

Simulation execution begins with the following command:

```
simulation.addOnClockTickListener(this::updateProcessingforVms);
simulation.start();
```

Listing 7.11: *Start Simulation*

Output Results. Output tables are generated using the provided builders. These summarize VM execution and scheduling behavior:

```
new DynamicVmTableBuilder(finishedVms).build();

List<SpotInstance> finishedSpot = new ArrayList<>();
for (DynamicVm vm : finishedVms) {
  if (vm instanceof SpotInstance) {
    ((SpotInstance) vm).calculateAverageInterruptionTime();
    finishedSpot.add((SpotInstance) vm);
  }
```

```
9  }
10
11 new SpotVmTableBuilder(finishedSpot).build();
```

Listing 7.12: *Build Output Tables*

7.2 Implementation Test Cases

To illustrate how the new features integrate with CloudSim Plus and how they behave during simulation, two test cases were developed:
`RandomlyGeneratedInstances` and `RestartingInterruptedSpot`.

RandomlyGeneratedInstances. This scenario demonstrates the automatic interruption and termination of Spot Instances. The simulation setup includes the creation of a simulation object, a datacenter with hosts, a datacenter broker, and a set of initial virtual machines and cloudlets. These VMs and cloudlets are submitted to the broker.

Additionally, a `clockTickListener` (see Section 5.2) is added, which dynamically generates new VM instances and associated cloudlets during simulation runtime. These VMs may be either On-Demand or Spot Instances. If insufficient capacity is available to allocate an On-Demand instance, running Spot Instances will be terminated to free resources. The output of this simulation, shown in Figure 7.1, presents a list of VMs executed during the simulation, including their final states. Interrupted Spot Instances appear with the status `TERMINATED`. Portions of the code were adapted from the examples in the CloudSim Plus repository.

RestartingInterruptedSpot. This test case demonstrates persistent request behavior and the resubmission of interrupted Spot Instances. The simulation setup is similar to the previous example, with the addition that Spot Instances are created first and On-Demand Instances are added with a delay of 10 seconds.

To enable automatic resubmission, Spot Instances are configured with persistent requests, a maximum waiting time, and a hibernation time limit.
An `onHostDeallocationListener` (see Section 5.2) is used to trigger re-submission whenever a VM completes execution on any host. Alternatively, a `clockTickListener` could be used for periodic checks.

During the simulation, Spot Instances are preempted by On-Demand Instances and later resumed when resources become available. The output is shown in Figures 7.1 and 7.2. The first figure displays all VMs, while the second focuses on Spot Instances and includes their average interruption times.

7.3 Google Cluster Trace

To further assess the practical value of the implementation, the Spot Instance behavior was tested using real-world workload data from the Google Cluster Trace. CloudSim

SIMULATION RESULTS

Broker ID	Virtual Machine (VM) Identifier	DC ID	Host Identifier	Host PEs CPU cores	VM PEs CPU cores	Start Time Seconds	Stop Time Seconds	Delay Seconds	Type	State
2	6	1	1	8	4	10	32	10	On-Demand	FINISHED
2	5	1	0	8	4	10	32	10	On-Demand	FINISHED
2	7	1	0	8	4	32	54	0	On-Demand	FINISHED
2	0	1	1	8	4	32	43	0	Spot	FINISHED
2	2	1	0	8	4	32	43	0	Spot	FINISHED
2	1	1	1	8	4	32	43	0	Spot	FINISHED
2	4	1	0	8	4	10	32	10	On-Demand	FINISHED
2	3	1	1	8	4	10	32	10	On-Demand	FINISHED

Figure 7.1: *Example Output – `RestartingInterruptedSpot` (DynamicVmTableBuilder)*

SIMULATION RESULTS

Broker ID	Virtual Machine (VM) Identifier	DC ID	Host Identifier	Host PEs CPU cores	VM PEs CPU cores	Start Time Seconds	Stop Time Seconds	State	Average Interruption Seconds
2	0	1	1	8	4	0	43	FINISHED	22
2	2	1	0	8	4	0	43	FINISHED	22
2	1	1	1	8	4	0	43	FINISHED	22

Figure 7.2: *Example Output – `RestartingInterruptedSpot` (SpotVmTableBuilder)*

Plus provides a basic trace reader for this dataset, which was adapted and extended to incorporate the new Spot Instance functionality.

This section begins with an overview of Google's Borg cluster manager, followed by a description of the 2011 cluster trace. The example implementation is then discussed, and the results of the simulation are analyzed.

Google's Cluster Manager – Borg. Borg is Google's internal cluster management system. It orchestrates hundreds of thousands of jobs, each composed of multiple tasks, across many clusters. Each job is executed on a "cell," a logical unit consisting of approximately 10,000 physical machines interconnected by a high-performance network [73, p. 1].

Datacenters are composed of multiple clusters, which in turn comprise one or more cells. All machines within a cell have similar hardware configurations in terms of CPU, RAM, disk, and network capacity [73, p. 2]. Borg handles both high-priority services (e.g., Gmail, Google Docs) and low-priority batch jobs. While most batch jobs are preemptible, high-priority services tend to dominate resource usage, accounting for approximately 60% of CPU and 85% of memory consumption. Borg typically manages workloads directly on physical machines without virtualization, due to historical limitations in hardware support and to avoid virtualization overhead [73, p. 2].

7.3.1 Google Cluster Data

In May 2011, Google publicly released trace data from one of their production clusters. This trace spans 30 days and includes activity logs from a cell with approximately 12,600

machines [74, p. 1]. The primary goal of the release was to facilitate academic research on cluster scheduling complexity [75, p. 2]. To preserve user privacy, the dataset applies various obfuscation techniques, such as hashing and numeric scaling.

The trace includes six tables, categorized into machine-related and job-related information. The *Machine Events* and *Machine Attributes* tables describe cluster infrastructure, including machine lifecycle events (e.g., addition, removal, resource updates) and static attributes (e.g., CPU frequency, kernel version) [75, p. 4].

The job and task-related tables are *Job Events*, *Task Events*, *Task Constraints*, and *Task Usage*. The job and task events track various lifecycle events: submission, scheduling, termination (due to eviction, failure, kill, etc.), and resource updates. Each event is timestamped and associated with unique identifiers for both jobs and tasks [75, p. 6].

Task Constraints specify requirements for task placement on machines, while the *Task Usage* table contains detailed resource usage statistics such as CPU, memory, and disk utilization [75, p. 9].

Analysis. The cluster data was recorded over a 30-day period, from May 1st, 2011, 19:00 to May 30th, 2011, 19:00. Due to this, full 24-hour data is not available for the first and last day. In total, *48,375,166 tasks* were submitted during the monitoring period. This number includes both original task submissions and resubmitted tasks that were evicted from machines. At any given time during the trace period, at least *97,145 tasks* were running concurrently across the cluster.

Figure 7.3 illustrates the range of concurrently running tasks per day. It shows the minimum and maximum number of tasks active at the same time on each day. The day with the lowest concurrency was Day 30, while the peak occurred on Day 25 with *223,355 concurrently active tasks*. Across the 30 days, the number of concurrently active tasks ranged on average from *114,023 to 157,442*.

The workload was distributed across *12,585 distinct host machines*. The most heavily loaded single host executed a total of *8,071 tasks* during the observed time period, and the maximum number of concurrently running tasks on a single host was *410*. Due to incomplete entries, *821,866 tasks* (approximately *1.7%*) could not be matched with a host and were excluded from detailed analysis.

Simulation-Specific Data. Because the Google trace does not use virtual machines—instead executing tasks directly on physical hosts—an additional step was required to adapt the data to CloudSim Plus. To enable simulation, it was assumed that all tasks submitted by the same user to the same host were executed within a single virtual machine. Accordingly, virtual machine IDs were synthesized from a combination of username and machine ID.

Using this assumption, the simulation generated *28,769,042 virtual machines*. At any point in the trace, a minimum of *75,947 virtual machines* were concurrently active, with a peak of *129,315 concurrent VMs* on Day 25. The maximum number of VMs active simultaneously on a single host was *25*. Daily averages ranged between *87,878 and 111,803 concurrent virtual machines*.

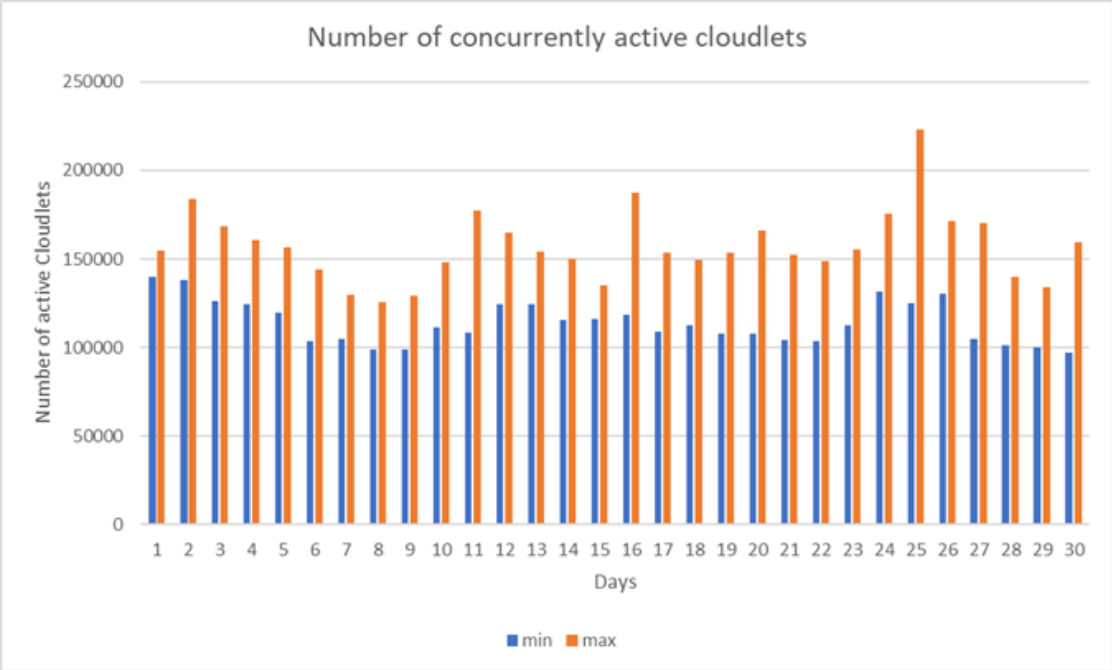

Figure 7.3: *Maximum and minimum number of concurrently active tasks per day*

7.3.2 Simulation with Cluster Data

The simulation incorporates two data tables from the Google Cluster Trace dataset: the *Machine Events* and the *Task Events* tables. The *Machine Events* data was used to create, submit, and remove hosts in the simulated datacenter, while the *Task Events* data was used to generate virtual machines and associated cloudlets.

Trace Reader. CloudSim Plus includes a trace reader capable of parsing data from the *Machine Events*, *Task Events*, and *Task Usage* tables. To enable a simulation that combines both machine and task events, and to reduce simulation time, several modifications were made to this reader.

In the original implementation, task entries with assigned machine IDs were not passed during event creation in the `GoogleTaskEventsTraceReader.java`. This was modified to support correct allocation of tasks to specific machines. Additionally, a new variable `cloudletHashMap` was introduced to accelerate lookup operations for cloudlets, significantly improving performance.

To further enhance performance, functionality related to dynamic resource updates was disabled, as all virtual machines in this simulation are of fixed size. The broker used in the trace reader was also changed from `DatacenterBrokerSimple` to the newly implemented `DatacenterBrokerDynamic` (see Section 5).

Changes were also made to the `TaskEventType` class, which processes events from the *Task Events* table. The primary modifications concerned the `SUBMIT` event. Previously,

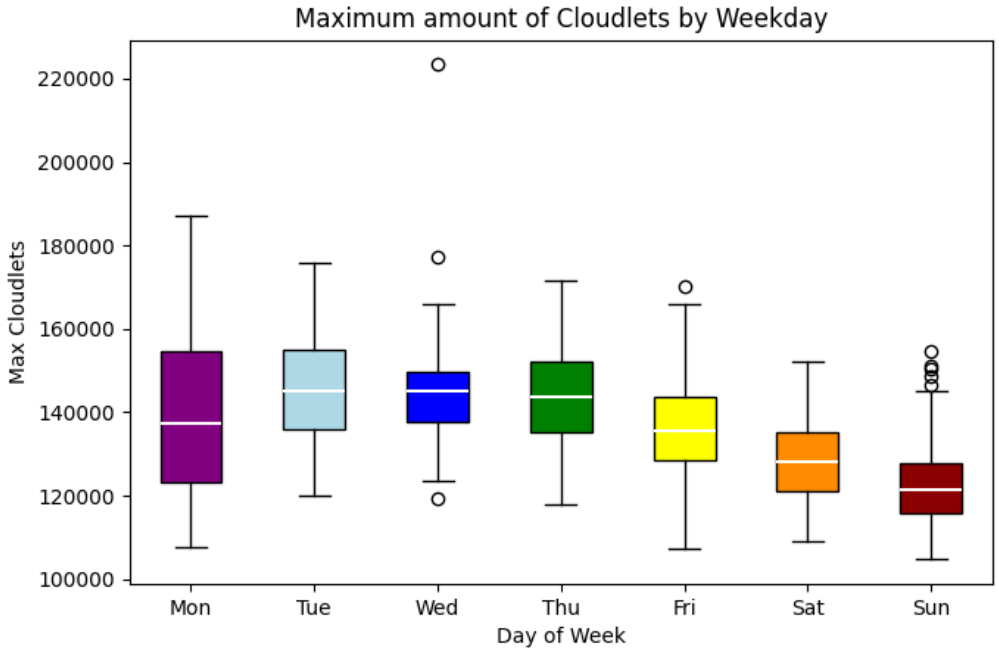

Figure 7.4: *Daily distribution of maximum concurrently running cloudlets (hourly resolution)*

cloudlets were created without checking for duplicate IDs. This was updated to check whether a cloudlet with the same ID already existed. If so, the existing ID was reassigned, and only the most recent cloudlet retained the original ID. All created cloudlets were added to the `cloudletHashMap`.

To minimize startup load, only cloudlets with a submission delay of zero are submitted at simulation start. Cloudlets with a delay are now added to a waiting list and submitted only when their execution time begins. This reduces simulation overhead and improves scalability.

The handling of `EVICT` and `FAIL` events was also revised. Previously, evicted cloudlets were paused and later resumed. In the updated implementation, evicted cloudlets are terminated and re-created upon rescheduling, as resubmitted tasks are typically scheduled to different machines. For `FAIL` events—which were previously ignored—cloudlets are now marked as finished to reflect their lifecycle in the simulation results.

The `cloudletLookup` method, used to identify cloudlets by ID, was optimized by switching from linear iteration over all cloudlets to a hash-based lookup using the `cloudletHashMap`.

Data Preparation. The two tables used required preprocessing to be compatible with the modified trace reader. The *Machine Events* table included many entries with missing CPU and RAM values. These gaps were filled by copying resource values from other machines. While this introduces inaccuracies, the deviation was deemed acceptable for

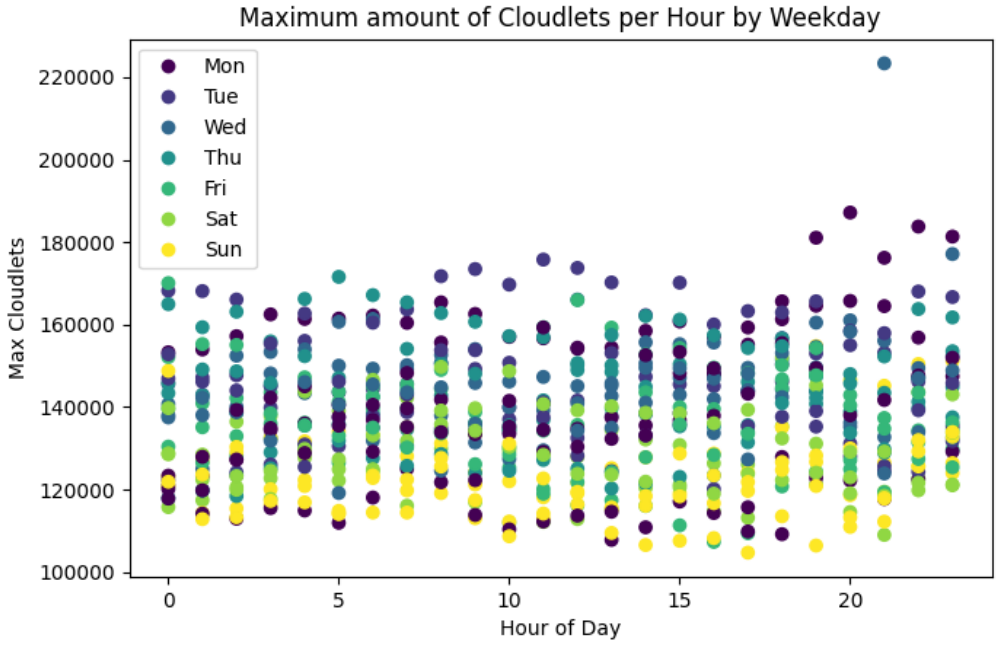

Figure 7.5: *Maximum number of concurrently running cloudlets by hour of the day*

the purpose of testing spot instance behavior.

In the *Task Events* table, some `SUBMIT` events lacked machine IDs. For these, subsequent events with the same job and task ID were checked. If a later event contained a machine ID, that value was copied into the `SUBMIT` event. If the next event was also a `SUBMIT`, the machine ID remained undefined. Tasks still lacking a machine ID after preprocessing were assigned to hosts based on the allocation policy.

7.4 Simulation Implementation & Output

CloudSim Plus provides two examples that use the trace reader: `GoogleMachineEventsExample1` and `GoogleTaskEventsExample1`, which respectively demonstrate how to create hosts and cloudlets from the Google Trace dataset (see Section 7.3.1). This simulation combines and extends those examples to generate both hosts and virtual machines, allocate tasks to those VMs, and run them on the specified machines from the dataset.

To evaluate spot instance functionality with realistic data, an additional 200,000 spot instances were submitted at the simulation start. In the one-day simulation scenario, each spot instance was configured to simulate an execution time of 20 hours. For the two-day simulation, their duration was set to 40 simulated hours. The simulated runtime was intentionally shorter than the actual simulation time (24 and 48 hours, respectively) to ensure that the spot instances could complete execution—even with interruptions and rescheduling—and avoid issues during shutdown.

CloudSim Plus normally updates cloudlet processing periodically throughout the simulation. However, doing so for millions of cloudlets from the Google trace would be computationally prohibitive. Since the finish times are known from the dataset, this periodic processing was disabled. For spot instances, cloudlet processing was manually updated within the cluster trace simulation logic. This change allowed the simulation to complete a one-day trace in under two days of real time. By further optimizing processing and only updating spot cloudlets after the full 20-hour duration, the simulation time was reduced to a few hours.

The real cluster trace shows that up to 21 virtual machines could be concurrently active on a single host. Because resource usage data for individual tasks was not available or usable at scale, fixed sizes were assigned to all hosts and VMs. Host resource sizes were based on the machine events file, while VM sizes were chosen to allow slightly more than 21 VMs per host. This ensured that capacity limitations would lead to interruptions during simulation. Although this reduces fidelity compared to the original cluster environment, the missing machine event data (see Section 7.3.2) already limits the accuracy of available capacity information.

Simulation Output. The simulation produces four CSV files as output. Two of these files contain data derived from the Google Cluster trace: one listing all successfully executed cloudlets, and the other listing all virtual machines that completed their assigned tasks. These files are generated using the `DynamicVmTableBuilder`.

The remaining two files contain information specific to the spot instances. One details all spot cloudlets, and the other lists all spot VMs, including their interruption history, generated by the `SpotVmTableBuilder`. In addition, a JSON file is generated to record the execution history of each spot instance that was interrupted and later resumed. Only interrupted instances are included in this file.

Finally, the entire console output of the simulation is saved to a separate text file for reference and debugging.

7.4.1 Simulation Performance

The simulation was executed on a Google Cloud virtual machine of type `e2-highmem-4`, a general-purpose instance with increased memory capacity. This machine provides 4 vCPUs and 32 GB of RAM.

To estimate the feasibility of simulating the full dataset with the available resources, initial simulation attempts were performed using only a subset of the Google Trace data. Simulating approximately one day of trace data, corresponding to the first 18 task event files (`part-00000-of-00500.csv` to `part-00017-of-00500.csv`), took 139,744.54 seconds, which equals 1 day, 14 hours, 49 minutes, and 4.54 seconds.

In this setup, virtual machines derived from the trace were allocated to the specific hosts referenced in the dataset, which helped reduce allocation overhead. The spot instance allocation used the `WorstFitVmAllocationPolicy`, one of the standard algorithms provided by CloudSim Plus. To test whether the allocation policy significantly impacted

performance, the simulation was rerun with the `FirstFitVmAllocationPolicy`, resulting in a total execution time of 134,087.57 seconds—roughly 1.5 hours faster.

During execution, average CPU utilization was approximately 28%, with around 1.15 cores actively used. Memory utilization remained steady at approximately 20%, or about 8.3 GB. Figures 7.6 and 7.7 show the resource usage over time. These results suggest that the simulation did not fully utilize the available resources, and thus, increasing CPU or memory alone would likely not improve performance.

Figure 7.6: *CPU Utilization Simulation*

Figure 7.7: *Memory Utilization Simulation*

When attempting to simulate multiple days of trace data with otherwise unchanged parameters, the simulation time increased exponentially. One such attempt ran for over 20 days without completing, progressing only to about 25% of the simulation (35,622.49 seconds of simulated time, or roughly half a day of the intended two-day window). Despite the long runtime, resource utilization remained low, reinforcing the conclusion that computational resources were not the limiting factor.

These findings suggest that the primary performance bottleneck in CloudSim Plus for large-scale simulations is not the virtual machine allocation process, but rather the overhead of processing cloudlet execution across a vast number of virtual machines. A potential solution would be to parallelize cloudlet processing updates across multiple threads or processes.

7.4.2 Final Simulation & Results

In the final simulation run, the first two days of the Google Trace dataset were used. This covered the first 36 task event files (`part-00000-of-00500.csv` to `part-00035-of-00500.csv`). The simulation took 599,006.12 seconds to complete, which equals approximately 166.3 hours or roughly 7 days.

Despite optimizations—such as skipping internal cloudlet processing for trace data and setting fixed durations for spot instances—the simulation still exhibited exponential runtime growth. Similar to prior attempts, CPU and memory usage remained below 30% and 25% respectively, further supporting the conclusion that performance was constrained by simulation logic rather than hardware limitations.

The simulation created a total of 2,384,409 virtual machines from the trace data. In addition, 200,000 spot instances were submitted at the start of the simulation, each with a fixed execution duration of 144,000,000 simulation units (equivalent to 40 hours of simulated time).

Spot Instance Interruption Statistics:

- **16.54%** of spot instances completed execution without interruption.
- **166,918** spot instances were interrupted at least once.
- **92,554** of the interrupted instances were redeployed.
- Of those:
  - **86,789** were redeployed once.
  - **5,693** were redeployed twice.
  - **72** were redeployed three times.
  - **0** instances were redeployed more than three times.
- The average interruption time for spot instances was **1,910.06 seconds**.
- The shortest interruption recorded was **37.81 seconds**, and the longest was **7,711.25 seconds** (approx. 2.14 hours).
- The maximum hibernation time allowed was 18,000 seconds (5 hours). Instances exceeding this threshold were terminated.

Completion Statistics:

- **76,960** spot instances (38.48%) finished execution.
- Of these, **43,878** had experienced at least one interruption and were later resumed.
- **123,040** spot instances were ultimately terminated.

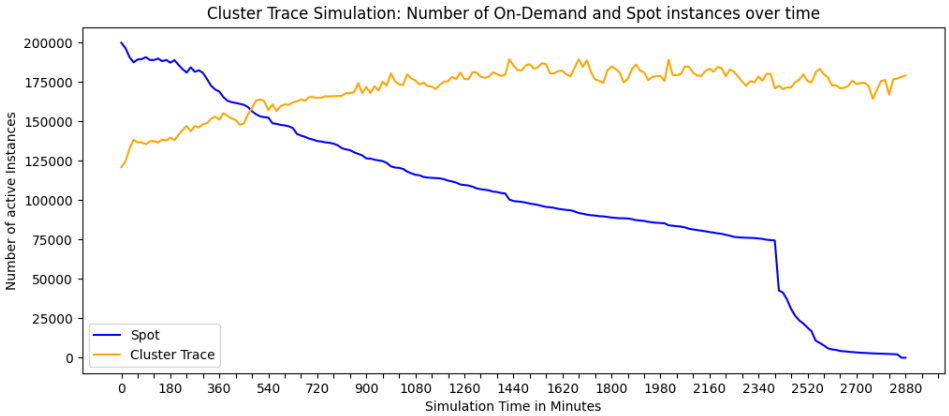

Figure 7.8: *Cluster Trace Simulation: Active virtual machine instances over time*

- **74,364** were terminated without any redeployment.
- **48,676** were terminated after being redeployed at least once.

Figure 7.8 shows the number of active virtual machines over time during the simulation. The orange line represents VMs from the cluster trace, while the blue line shows active spot instances. Data points were recorded at 15-minute intervals. As expected, the increase in trace VMs led to more frequent interruptions of spot instances. Occasionally, a dip in cluster VMs corresponds to a rise in active spot instances—though shorter periods of redeployment may not be visible due to the sampling interval. The sharp decline in spot instances at the 2,400-minute mark (40 hours) reflects the fixed execution time reaching its limit.

As noted in Section 7.3.1, day two of the trace has one of the highest VM counts. Based on this trend, simulating later days—where the number of cluster VMs decreases—would likely result in a higher number of uninterrupted spot instances.

7.5 Comparison of Allocation Algorithms

To evaluate the performance and effectiveness of the implemented HLEM-VMP algorithm and its adjusted version, a comparative analysis was conducted using a sample simulation. Additionally, both algorithms were compared with the First-Fit algorithm, which is already included in the CloudSim Plus framework.

7.5.1 Evaluation Criteria

The evaluation was based on the following key metrics:

- Number of spot instance interruptions
- Average interruption time of spot instances

7.5.2 Simulation Setup

Table 7.1 presents the host types used in the simulation. Four host configurations—*small*, *medium*, *large*, and *x-large*—were defined, with each successive type offering increased CPU, memory, bandwidth, and storage capacities. The simulation included 20 small, 30 medium, 30 large, and 20 x-large hosts.

Virtual machines were defined using 10 distinct VM profiles, each with varying resource demands in terms of CPU, memory, bandwidth, and storage. These profiles are detailed in Table 7.2. Randomized values were used for both VM submission delays and total execution times. The same randomized values were reused across all simulation runs to ensure consistency.

A total of 2,000 VMs were submitted: 400 spot VMs and 600 on-demand VMs were submitted immediately, without delay. The remaining 1,000 VMs (50% of the total) were submitted with randomized delays. All randomized variables, including instance type, VM profile, delay, and execution duration, were kept identical across all algorithms to enable a fair comparison.

The scheduling algorithms evaluated in this section are the *HLEM-VMP* (Section 6.2), the *adjusted HLEM-VMP* (Section 6.3), and the *First-Fit* baseline provided by CloudSim Plus.

Table 7.1: *Host Types*

Size	**CPU**	**Memory**	**Bandwidth**	**Storage**
Small	8	16,384	5,000	200,000
Medium	16	32,768	10,000	400,000
Large	32	65,536	20,000	800,000
X-Large	64	131,072	40,000	1,600,000

Table 7.2: *VM Profiles*

ID	**CPU**	**Memory**	**Bandwidth**	**Storage**	**Spot Count**	**On-Demand Count**
#1	1	1,024	100	10,000	31	160
#2	2	1,024	100	10,000	42	175
#3	1	2,048	200	20,000	36	168
#4	2	2,048	200	20,000	44	146
#5	4	2,048	200	20,000	40	158
#6	4	4,096	500	50,000	40	145
#7	6	4,096	500	50,000	36	170
#8	6	8,192	1,000	80,000	51	155
#9	8	8,192	1,000	80,000	33	162
#10	10	8,192	1,000	80,000	47	168

7.5.3 Results and Analysis

This subsection presents the simulation results and highlights performance differences among the algorithms. Figure 7.9 displays the number of active spot and on-demand VMs over time. While on-demand activity remained nearly identical across all algorithms, differences emerged in the distribution of spot instances.

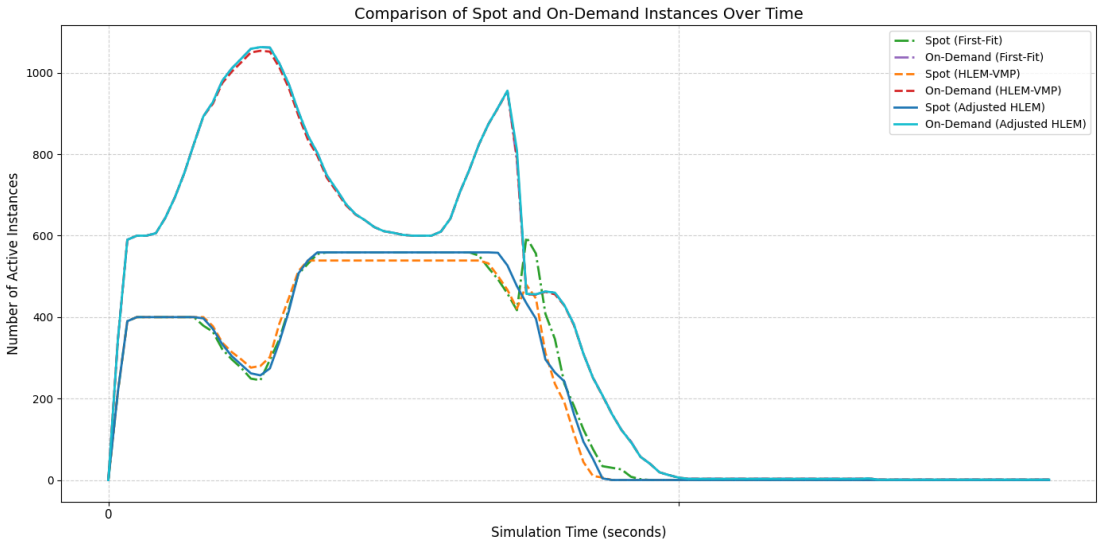

Figure 7.9: *Active Instances Over Time*

Figure 7.10 shows the total number of spot instance interruptions. The First-Fit algorithm resulted in the most interruptions (286), followed by HLEM-VMP (230), and the adjusted HLEM-VMP performed best with only 205 interruptions.

In all scenarios, the maximum number of interruptions per VM was two. Average interruption times are shown in Figure 7.11. HLEM-VMP achieved the shortest average interruption time (21.12 seconds), followed by First-Fit (22.81 seconds), while the adjusted HLEM-VMP had the highest (25.20 seconds). However, when comparing maximum interruption durations, the adjusted HLEM-VMP performed best (45.65 seconds), followed by HLEM-VMP (49.49 seconds), and First-Fit (64.87 seconds). Minimum interruption times were nearly identical across all algorithms.

Interruption statistics were gathered using the `SpotVmTableBuilder` for aggregated interruption data and the `ExecutionTableBuilder` for execution timelines. These results confirm that allocation strategies can meaningfully affect spot instance behavior and demonstrate that adjustments to HLEM-VMP can reduce interruption frequency in dynamic cloud simulations.

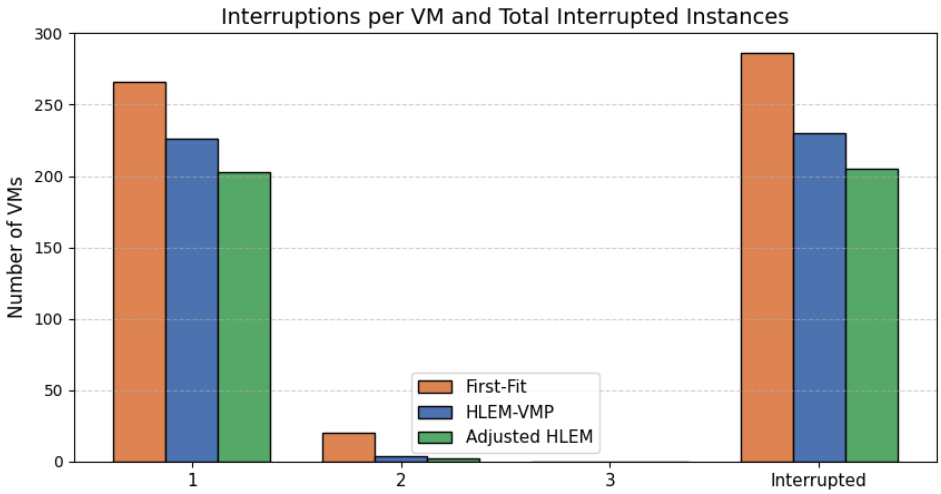

Figure 7.10: *Total Spot Instance Interruptions*

7.6 Outlook: Interruption Frequency Data

At the time of writing, the exact criteria cloud providers such as AWS use to determine which spot instances are interrupted remains undisclosed. This raises the question of whether certain instance attributes influence the likelihood of interruption.

To provide some visibility into this behavior, AWS offers the Spot Instance Advisor[2], which categorizes interruption frequency into five predefined ranges. A screenshot of the Spot Instance Advisor interface is shown in Figure 12. However, the data does not provide precise interruption probabilities, only the following broad categories:

- Less than 5%
- Between 5% and 10%
- Between 10% and 15%
- Between 15% and 20%
- Greater than 20%

Although this format offers general insight, it lacks granularity—especially for instances in the highest range, where interruption frequency may vary significantly.

Along with these interruption estimates, the Spot Advisor dataset includes additional attributes for each instance: instance type category (e.g., general purpose, compute optimized), vCPU count, memory size (in GiB), and expected savings compared to

[2] `https://spot-bid-advisor.s3.amazonaws.com/spot-advisor-data.json`

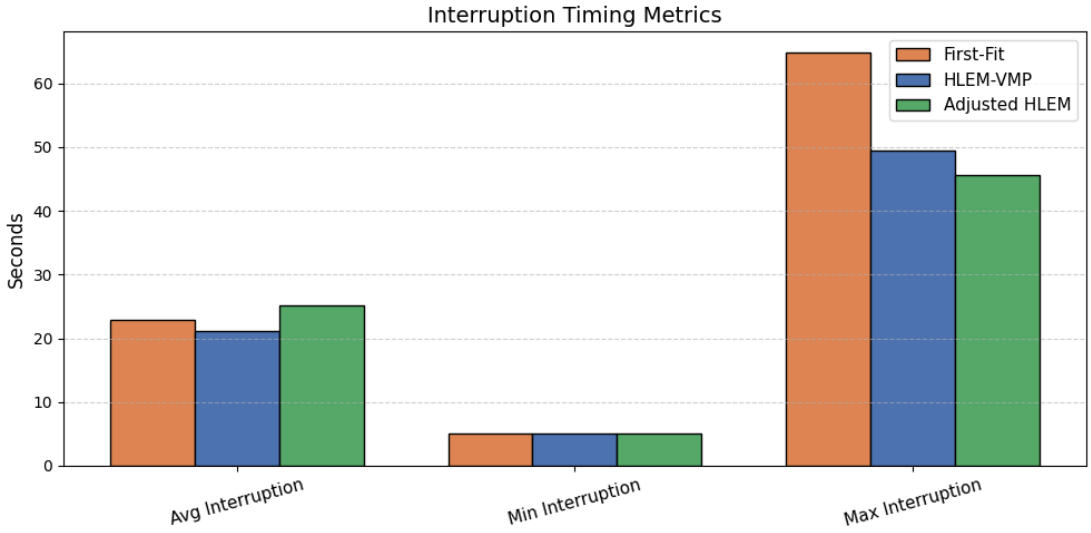

Figure 7.11: *Spot Instance Interruption Durations*

on-demand pricing. The data is also region-specific (e.g., US East – N. Virginia, Europe – Frankfurt), and is further divided by operating system (Linux/Unix or Windows). In total, the dataset includes 389 instance types, although availability varies across regions.

To supplement this dataset, spot price data from the AWS Spot Price feed[3] was merged into the dataset based on region and instance type.

To enrich the feature space, additional instance metadata was obtained from the AWS Console[4], which provides per-region CSV files listing detailed specifications. This data includes the number of availability zones in which the instance is offered, number of GPUs, whether the instance belongs to the current generation, clock speed in GHz, storage type (SSD, HDD, or none), storage capacity in GB, network performance class, number of physical cores, instance family, and on-demand pricing for both Linux and Windows operating systems.

To support analysis, instances were categorized hierarchically by their broader category (e.g., general purpose), instance family (e.g., a1), and exact machine type (e.g., a1.xlarge). Instances within the same family differ primarily in the number of CPUs, memory size, and storage configuration.

Data Preparation. To prepare the data for analysis, several Python scripts were developed to automate data collection and integration. The dataset was saved in two formats: a CSV file for statistical processing and a JSON file for human readability and documentation of transformation steps.

To account for temporal variation in spot prices and availability, data collection was

[3] `https://spot-price.s3.amazonaws.com/spot.js`
[4] `https://console.aws.amazon.com/ec2/`

repeated over multiple days at randomized times. The integration process involved the following steps:

1. Retrieving Spot Advisor data and loading it into Python as a dictionary.
2. Downloading console data for each AWS region and extracting relevant attributes.
3. Retrieving spot price data and merging it based on instance type and region.

The dataset was preprocessed to support quantitative analysis. String-based numeric fields (e.g., prices) were converted to floating-point numbers, and categorical features were transformed into numerical labels using encoding techniques. Additional derived features were created, such as the price per gigabyte of RAM and the price per physical core.

The resulting dataset offers a comprehensive set of features for evaluating the relationship between instance characteristics and interruption frequency. Table 7.3 present excerpts from the first five rows of the final CSV dataset used in the analysis.

Feature Analysis and Selection. To evaluate the relationship between various instance features and the frequency of interruption, a correlation matrix was generated using the `associations()` function from the `dython.nominal` library. This method is well-suited for mixed-type datasets, as it applies appropriate correlation measures depending on the feature types:

- **Theil's U** for nominal–nominal associations (e.g., *region* vs. *operating system*)
- **Correlation Ratio (η^2)** for numeric–categorical relationships (e.g., *vCPU count* vs. *interruption frequency*)
- **Pearson correlation** for numeric–numeric comparisons

This approach allows all feature types to be included in a unified heatmap representation (see Figure 7.12).

Among the nominal features, the strongest associations with interruption frequency were:

- **Instance type** (`instance_type_list`): Correlation = 0.38
- **Instance family** (`instance_family`): Correlation = 0.33
- **Machine type** (`machine_type`): Correlation = 0.18

These findings suggest that more specific classifications (e.g., concrete instance type) have stronger predictive power for interruption behavior than broader categories.

In contrast, features such as *day*, *free_tier*, *free_trial*, and *dedicated_host* showed negligible correlation and were excluded from further analysis.

To complement this analysis and validate the findings, the following methods are proposed for future evaluation:

- **Mutual Information (MI):** To detect non-linear dependencies between features and interruption frequency.
- **Tree-based feature importance:** Random Forest classifiers to rank the most influential predictors.
- **ANOVA and Chi-square tests:** For evaluating associations between continuous and categorical features.

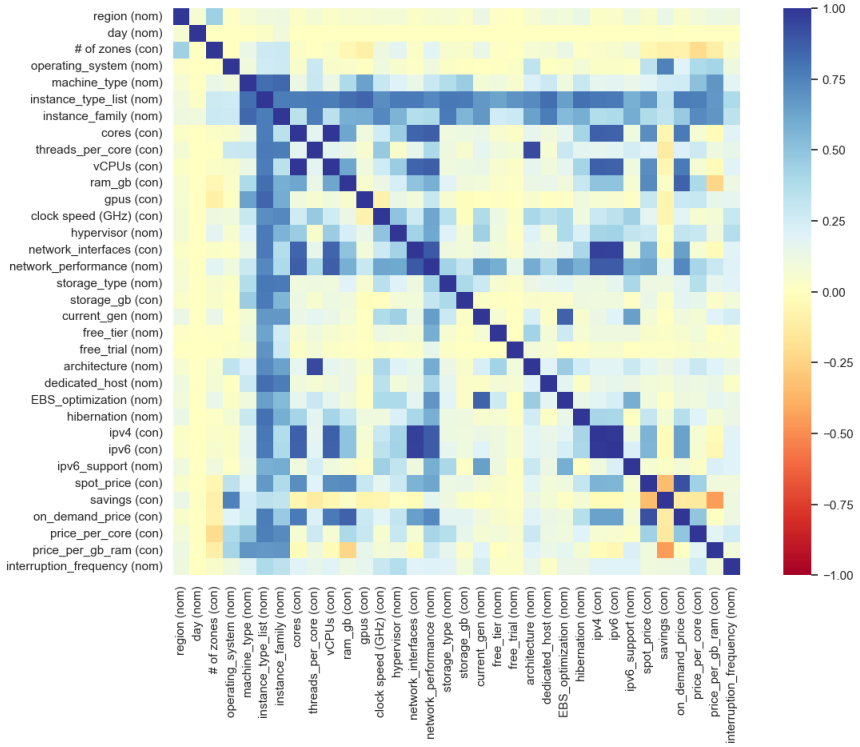

Figure 7.12: *Correlation Matrix Frequency Analysis*

Table 7.3: *Selected Features for Interruption Frequency Analysis (Sample Entries)*

(a) *Hardware and Instance Configuration*

#	Zones	Arch.	GHz	Cores	Gen	Host	Free Tier	Free Trial	GPUs	IF
1	3	x86_64	3.1	64	Y	Y	N/A	N/A	0	2
2	3	x86_64	3.1	48	Y	Y	N/A	N/A	0	3
3	3	x86_64	3.4	4	Y	Y	N/A	N/A	0	0
4	3	x86_64	3.6	96	Y	N	N/A	N/A	0	2
5	3	x86_64	3.1	4	Y	Y	N/A	N/A	0	1

(b) *Network, Pricing, and Resources*

#	Net Perf.	On-Demand	$Core	$GB RAM	RAM (GB)	OS	IF
1	20.0	4.048	0.06325	0.01581	256.0	Linux	2
2	50.0	7.128	0.14850	0.01856	384.0	Linux	3
3	9.0	0.213	0.05325	0.02662	8.0	Linux	0
4	25.0	5.861	0.06105	0.03053	192.0	Linux	2
5	9.0	0.299	0.07475	0.01869	16.0	Linux	1

(c) *Region, Storage, and Instance Metadata*

#	Region	Savings (%)	Spot Price	Storage (GB)	Type	vCPUs	IF
1	eu-central-2	73	1.076	0	m5.16xlarge	64	2
2	eu-central-2	70	2.1384	30000	i3en.12xlarge	48	3
3	eu-central-2	70	0.0641	0	c5.xlarge	4	0
4	eu-central-2	74	1.5372	3600	c5d.metal	96	2
5	eu-central-2	77	0.0673	150	m5d.xlarge	4	1

Notes: IF = Interruption Frequency (0–4). "Y" indicates feature is supported; "N" = not supported; "N/A" = not available.

8 Limitations

This section outlines the boundaries of this thesis, focusing on the design decisions and areas that were intentionally not pursued to maintain a clear and achievable scope. These decisions reflect a strategic focus on extending an existing cloud simulation framework with support for spot instances, rather than building a general-purpose simulation or conducting an exhaustive data science analysis.

Simulation Scope and Practical Constraints

Due to the resource-intensive nature of large-scale simulation, and in alignment with the available infrastructure (a Google Cloud virtual machine under a free-tier trial), only a portion of the cluster data was used. This choice allowed for validating the simulation extension under realistic conditions, while keeping resource consumption within practical limits. Certain optimizations—such as skipping unnecessary cloudlet runtime calculations—were deliberately introduced to make this feasible.

Focus on Simulation Extension, Not Market Modeling

While the thesis references aspects of real-world cloud markets, it does not attempt to model cloud business dynamics (e.g., pricing strategies, provider bidding mechanisms, or SLA-based penalties). These topics are both highly complex and context-specific, and incorporating them would have significantly expanded the scope beyond the intended technical focus on infrastructure-level simulation features.

Feature Analysis and Data Limitations

The interruption frequency analysis of AWS spot instances was based on publicly available snapshot data from the Spot Instance Advisor. This approach was chosen for its simplicity and accessibility, and served well for a preliminary investigation. However, due to the static nature of the dataset, it cannot account for temporal variations such as weekday-based fluctuations or longer-term trends in instance availability. Collecting time-series data over an extended period would allow for more accurate modeling, but was outside the scope of this work.

Data Science Techniques

Basic data science techniques were applied during the feature analysis phase, supported by self-directed learning and prior coursework. While more advanced methods may offer improvements, the intent of this section was exploratory: to identify potential correlations

between instance features and interruption likelihood. As such, the level of depth chosen here was appropriate for the thesis objective, which remained focused on simulation design and implementation.

Framework Constraints and Design Choices

The decision to use CloudSim Plus was based on its maturity, modular design, and active maintenance. Developing a simulation framework from scratch was considered out of scope due to the significant time and effort required. Working within an existing framework also provided the advantage of leveraging tested and validated components.

At the same time, this choice introduced some structural limitations. For example, certain core framework classes and variables are not easily accessible or modifiable, which required careful architectural planning and sometimes constrained the design space. Nonetheless, these limitations were accepted in favor of maintaining compatibility and avoiding fragile modifications to the core framework.

9 Conclusion

The widespread adoption of cloud platforms across diverse application domains has led to increasing interest in optimizing key aspects such as resource allocation, scheduling, energy efficiency, and virtual machine migration policies. However, conducting experiments on public cloud infrastructure is often associated with high operational costs and limited control over internal behaviors. Cloud simulation provides a cost-effective and scalable alternative, enabling researchers to evaluate new strategies under controlled conditions.

Among the available frameworks, CloudSim and CloudSim Plus are widely used in academic research. Despite their flexibility, these frameworks lack support for accurately modeling spot (or preemptible) instances—VMs offered at significant discounts by cloud providers to utilize spare capacity, but which can be interrupted at any time. Understanding the behavior of such instances is critical for developing effective, cost-saving strategies in Infrastructure-as-a-Service environments.

This thesis addressed this limitation by designing and implementing an extension to CloudSim Plus that enables realistic simulation of spot instance behavior. The extension models the full spot instance lifecycle, including the interruption of instances to free capacity for on-demand requests and configurable interruption strategies such as termination and hibernation. Furthermore, support for dynamic provisioning was introduced, allowing both spot and on-demand VMs to be resubmitted if their initial placement fails due to insufficient resources.

As a proof of concept, a simulation based on the Google Cluster Trace dataset was conducted. Rather than attempting to replicate the full trace, the simulation focused on demonstrating the feasibility and effectiveness of modeling dynamic spot instance behavior—specifically the hibernation and resumption of instances in response to resource availability. The scenario validates the ability of the extended framework to capture essential dynamics observed in real-world cloud environments.

To complement the modeling of spot instance behavior in CloudSim Plus, an allocation algorithm from existing literature—HLEM-VMP—was implemented. This heuristic-based algorithm utilizes host filtering and selection mechanisms to determine the most suitable host for a VM, with the objective of reducing SLA violations. It was chosen for its simplicity and its ability to support dynamic resource allocation without the overhead and complexity associated with meta-heuristic approaches. The primary input parameters for the algorithm are the resource demands of the virtual machines and the available capacities on the hosts. The purpose of integrating this algorithm was to demonstrate how the developed extension enables the evaluation and comparison of different VM allocation strategies. To that end, the algorithm was further refined to reduce the number of interruptions by factoring in the proportion of host resources currently consumed by spot instances in the host selection score calculation. The adjusted HLEM algorithm

was evaluated against the original HLEM-VMP and the First-Fit strategy. Results demonstrated that the adjusted version could reduce the number and maximum duration of spot instance interruptions under certain conditions. However, this improvement came at the cost of a slightly increased average interruption time, highlighting a trade-off between frequency and duration of interruptions.

Currently, the selection of which spot instances to terminate during resource contention is determined in a non-deterministic manner, based solely on the VM list associated with a host. This introduces potential inefficiencies. Future research could explore the implementation of targeted deallocation strategies or predictive models to more intelligently select interruption candidates based on historical patterns, VM importance, or runtime behavior. Additionally, a preliminary analysis was conducted to examine which instance characteristics correlate with spot instance interruption frequencies, using publicly available data from the AWS Spot Instance Advisor. Further research in this direction—supported by empirical data—could contribute to the refinement of simulation models and allocation strategies, ultimately improving the accuracy of spot instance behavior modeling, particularly in relation to termination patterns.

In summary, this thesis contributes a meaningful extension to CloudSim Plus for simulating dynamic spot instance behavior and demonstrates its applicability in evaluating VM placement strategies. The proposed enhancements lay the groundwork for further research into cost-efficient, interruption-aware scheduling algorithms in cloud environments.

Bibliography

[1] K. A. Scarfone, M. P. Souppaya, and P. Hoffman, "Sp 800-125. guide to security for full virtualization technologies," 2011.

[2] A. Bhardwaj and C. R. Krishna, "Virtualization in cloud computing: Moving from hypervisor to containerization—a survey," *Arabian Journal for Science and Engineering*, pp. 1–17, 2021.

[3] A. M. Law, *Simulation Modeling and Analysis*, 5th ed. McGraw-Hill, 2013.

[4] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[5] T.-P. Pham, S. Ristov, and T. Fahringer, "Performance and behavior characterization of amazon ec2 spot instances," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 73–81.

[6] J. Surbiryala and C. Rong, "Cloud computing: History and overview," in *2019 IEEE Cloud Summit*. IEEE, 2019, pp. 1–7.

[7] V. Arutyunov, "Cloud computing: Its history of development, modern state, and future considerations," *Scientific and Technical Information Processing*, vol. 39, no. 3, pp. 173–178, 2012.

[8] S. Namasudra, P. Roy, and B. Balusamy, "Cloud computing: fundamentals and research issues," in *2017 Second International Conference on Recent Trends and Challenges in Computational Models (ICRTCCM)*. IEEE, 2017, pp. 7–12.

[9] N. Antonopoulos and L. Gillam, *Cloud computing*, 2nd ed. Springer, 2017.

[10] "Cloud infrastructure market," https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/, accessed: 2020-05-16.

[11] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011.

[12] A. Singh and K. Chatterjee, "Cloud security issues and challenges: A survey," *Journal of Network and Computer Applications*, vol. 79, pp. 88–115, 2017.

[13] U. U. Rahman, K. Bilal, A. Erbad, O. Khalid, and S. U. Khan, "Nutshell—simulation toolkit for modeling data center networks and cloud computing," *IEEE Access*, vol. 7, pp. 19 922–19 942, 2019.

[14] "Gartner cloud forecast," https://www.gartner.com/en/newsroom/press-releases/2021-04-21-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-23-percent-in-2021, accessed: 2021-05-16.

[15] B. Asvija, R. Eswari, and M. Bijoy, "Security in hardware assisted virtualization for cloud computing—state of the art issues and challenges," *Computer Networks*, vol. 151, pp. 68–92, 2019.

[16] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: state of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2017.

[17] F. SIERRA-ARRIAGA, R. BRANCO, and B. LEE, "Security issues and challenges for virtualization technologies," 2020.

[18] M. Portnoy, *Virtualization essentials*. John Wiley & Sons, 2012, vol. 19.

[19] A. Rashid and A. Chaturvedi, "Virtualization and its role in cloud computing environment," 2019.

[20] P. Kedia, R. Nagpal, and T. P. Singh, "A survey on virtualization service providers, security issues, tools and future trends," *International Journal of Computer Applications*, vol. 69, no. 24, 2013.

[21] L. Bouali, E. Abd-Elrahman, H. Afifi, S. Bouzefrane, and M. Daoui, "Virtualization techniques: Challenges and opportunities," in *International Conference on Mobile, Secure, and Programmable Networking*. Springer, 2016, pp. 49–62.

[22] "Virtualization a complete guide," https://www.ibm.com/cloud/learn/virtualization-a-complete-guide#toc-what-is-vi-yPgpyQPU, accessed: 2021-05-24.

[23] I. Alam, K. Sharif, F. Li, Z. Latif, M. Karim, S. Biswas, B. Nour, and Y. Wang, "A survey of network virtualization techniques for internet of things using sdn and nfv," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–40, 2020.

[24] V. G. da Silva, M. Kirikova, and G. Alksnis, "Containers for virtualization: An overview," *Applied Computer Systems*, vol. 23, no. 1, pp. 21–27, 2018.

[25] "Amazon aws documentation," https://docs.aws.amazon.com/index.html, accessed: 2020-05-16.

[26] A. Deldari and A. Salehan, "A survey on preemptible iaas cloud instances: challenges, issues, opportunities, and advantages," *Iran Journal of Computer Science*, pp. 1–24, 2020.

[27] "Google compute engine documentation," https://cloud.google.com/compute/docs, accessed: 2020-05-16.

[28] "Azure documentation," https://docs.microsoft.com/en-us/azure/?product=featured, accessed: 2020-05-16.

[29] P. Ambati, D. Irwin, and P. Shenoy, "No reservations: A first look at amazon's reserved instance marketplace," in *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[30] S. Yang, L. Pan, and S. Liu, "An online algorithm for selling your reserved iaas instances in amazon ec2 marketplace," in *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, 2019, pp. 296–303.

[31] G. George, R. Wolski, C. Krintz, and J. Brevik, "Analyzing aws spot instance pricing," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 222–228.

[32] M. Baughman, S. Caton, C. Haas, R. Chard, R. Wolski, I. Foster, and K. Chard, "Deconstructing the 2017 changes to aws spot market pricing," in *Proceedings of the 10th Workshop on Scientific Cloud Computing*, 2019, pp. 19–26.

[33] "Aws cloud console," https://console.aws.amazon.com, accessed: 2020-05-08.

[34] "Amazon compute service level agreement," https://aws.amazon.com/compute/sla/, accessed: 2020-05-08.

[35] "Amazon aws," https://aws.amazon.com, accessed: 2020-05-14.

[36] "Microsoft azure console," https://portal.azure.com/, accessed: 2020-05-15.

[37] "Sla für virtual machines," https://azure.microsoft.com/de-de/support/legal/sla/virtual-machines/v1_9/, accessed: 2020-05-11.

[38] "Google cloud console," https://console.cloud.google.com/, accessed: 2020-05-08.

[39] "Compute engine service level agreement," https://cloud.google.com/compute/sla, accessed: 2020-05-12.

[40] J. Byrne, S. Svorobej, K. M. Giannoutakis, D. Tzovaras, P. J. Byrne, P.-O. Östberg, A. Gourinovitch, and T. Lynn, "A review of cloud computing simulation platforms and related environments," in *International Conference on Cloud Computing and Services Science*, vol. 2. SCITEPRESS, 2017, pp. 679–691.

[41] P. J. Sanchez, "Fundamentals of simulation modeling," in *2007 Winter Simulation Conference*. IEEE, 2007, pp. 54–62.

[42] T. J. Schriber, D. T. Brunner, and J. S. Smith, "Inside discrete-event simulation software: How it works and why it matters," in *2017 Winter Simulation Conference (WSC)*. IEEE, 2017, pp. 735–749.

[43] W. K. V. Chan, Y.-J. Son, and C. M. Macal, "Agent-based simulation tutorial-simulation of emergent behavior and differences between agent-based simulation and discrete-event simulation," in *Proceedings of the 2010 winter simulation conference*. IEEE, 2010, pp. 135–150.

[44] L. Bertot, S. Genaud, and J. Gossa, "Improving cloud simulation using the monte-carlo method," in *European Conference on Parallel Processing*. Springer, 2018, pp. 404–416.

[45] "Cloudsim releases," https://github.com/Cloudslab/cloudsim/releases, accessed: 2021-05-07.

[46] H. Casanova, "Simgrid: A toolkit for the simulation of application scheduling," in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, 2001, pp. 430–437.

[47] H. Casanova, A. Legrand, and M. Quinson, "Simgrid: A generic framework for large-scale distributed experiments," in *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*. IEEE, 2008, pp. 126–131.

[48] "Virtualization / cloud abstractions in simgrid," https://simgrid.org/contrib/clouds-sg-doc.html, accessed: 2021-05-12.

[49] A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, and I. M. Llorente, "icancloud: A flexible and scalable cloud infrastructure simulator," *Journal of Grid Computing*, vol. 10, no. 1, pp. 185–209, 2012.

[50] I. K. Kim, W. Wang, and M. Humphrey, "Pics: A public iaas cloud simulator," in *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 2015, pp. 211–220.

[51] F. Fakhfakh, H. H. Kacem, and A. H. Kacem, "Simulation tools for cloud computing: A survey and comparative study," in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*. IEEE, 2017, pp. 221–226.

[52] M. Masdari, S. S. Nabavi, and V. Ahmadi, "An overview of virtual machine placement schemes in cloud computing," *Journal of Network and Computer Applications*, vol. 66, pp. 106–127, 2016.

[53] Z. Á. Mann, "Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms," *Acm Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–34, 2015.

[54] M. A. Kaaouache and S. Bouamama, "Solving bin packing problem with a hybrid genetic algorithm for vm placement in cloud," *Procedia Computer Science*, vol. 60, pp. 1061–1069, 2015.

[55] H. Feng, H. Li, Y. Liu, K. Cao, and X. Zhou, "A novel virtual machine placement algorithm based on grey wolf optimization," *Journal of Cloud Computing*, vol. 14, no. 1, pp. 1–16, 2025.

[56] R. K. Gupta and R. Pateriya, "Survey on virtual machine placement techniques in cloud computing environment," *International Journal on Cloud Computing: Services and Architecture (IJCCSA)*, vol. 4, no. 4, pp. 1–7, 2014.

[57] "Cloudsimplus website," https://cloudsimplus.org/, accessed: 2021-05-07.

[58] W. Attaoui and E. Sabir, "Multi-criteria virtual machine placement in cloud computing environments: a literature review," in *2024 International Conference on Ubiquitous Networking (UNet)*, vol. 10. IEEE, 2024, pp. 1–11.

[59] J. A. Murali and T. Brindha, "A meta-heuristic clustered grey wolf optimization algorithm for cloud resource scheduling," *International Journal of Advanced Technology and Engineering Exploration*, vol. 10, no. 107, p. 1336, 2023.

[60] S. Mehta, P. Kaur, and P. Agarwal, "Improved whale optimization variants for sla-compliant placement of virtual machines in cloud data centers," *Multimedia Tools and Applications*, vol. 83, no. 1, pp. 149–171, 2024.

[61] H. T. Ciptaningtyas, A. M. Shiddiqi, and D. Purwitasari, "A systematic literature review of genetic algorithm-based approaches for cloud task scheduling," in *2023 14th International Conference on Information & Communication Technology and System (ICTS)*, 2023, pp. 319–324.

[62] M. C. Çavdar, I. Korpeoglu, and Ö. Ulusoy, "A utilization based genetic algorithm for virtual machine placement in cloud systems," *Computer Communications*, vol. 214, pp. 136–148, 2024.

[63] Y. Jing, L. Shen, C. Yao, F. Fan, and X. Wang, "Hlem-vmp: An effective virtual machine placement algorithm for minimizing sla violations in cloud data centers," *IEEE Systems Journal*, vol. 18, no. 4, pp. 1963–1974, 2024.

[64] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf, "Nist cloud computing reference architecture," *NIST special publication*, vol. 500, no. 2011, pp. 1–28, 2011.

[65] "Google data centers," https://www.google.com/about/datacenters, accessed: 2021-09-15.

[66] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures," *Computer*, vol. 45, no. 12, pp. 65–72, 2012.

[67] M. C. Silva Filho, C. C. Monteiro, P. R. Inácio, and M. M. Freire, "Approaches for optimizing virtual machine placement and migration in cloud environments: A survey," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 222–250, 2018.

[68] "Spot instance advisor," https://aws.amazon.com/ec2/spot/instance-advisor/, accessed: 2021-03-18.

[69] M. C. Silva Filho, R. L. Oliveira, C. C. Monteiro, P. R. Inácio, and M. M. Freire, "Cloudsim plus: a cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 400–406.

[70] "Cloudsimplus documentation," https://cloudsimplus.readthedocs.io/en/latest/, accessed: 2021-05-07.

[71] "Google cluster data," https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md, accessed: 2021-05-07.

[72] K. Ettikyala and Y. R. Devi, "A study on cloud simulation tools," *International Journal of Computer Applications*, vol. 115, no. 14, 2015.

[73] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[74] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–14.

[75] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.